FUNCTIONAL PROGRAMMING AND THE LANGUAGE TALE

Henk Barendregt *
Marc van Leeuwen *

Mathematical Institute
Budapestlaan 6
3508 TA Utrecht
The Netherlands

Abstract. Chapter I is an introduction to functional programming, using informal (though precise) programs. Moreover, it is an introduction to chapter II, in which the functional language TALE is defined in a formal way. The definition of TALE consists of a core language based on the second order typed lambda calculus with recursive types and data types for tuples, unions and arrays. The semantics of the core language is given as a precisely defined reduction relation on expressions. The core language is extended in order to make programming more flexible. It is indicated how expressions in the extended language have to be translated to the core language. All purely functional languages are essentially based on the lambda calculus. Therefore TALE has a strong similarity to other such languges like SASL (Turner [1979a]), Miranda (Turner [1985]), LML (Augustsson [1984])and HOPE (Burstall et al [1980]). Our motivation for introducing TALE was to give a complete description of a language with precise semantics and at the same time a flexible syntax for actual programming.

* The authors are partially sponsored by the Dutch Ministery of Science and Education through the project "Parallel Reduction Machine". Chapter I is written by the first and II by the second author.

CONTENTS

CHAPTER I

INFORMAL FUNCTIONAL PROGRAMS

I.1 THE BASIC IDEA

I.1.1 Some History

The von Neumann computer with its imperative style of programming is based on a computation model introduced in 1936: the Turing machine. In that same year A. Church introduced a different model: the lambda calculus. A. Turing showed that the computational capacity of both models is the same. Functional programming is based on the second computational model. Reduction machines are being designed in order to evaluate functional programs.

Although conceived in the same year, the functional model of computation has gained popularity much slower than the imperative one. This is because the imperative model is much easier to implement on hardware. Nevertheless the functional model has been put foreward as an alternative by an increasing number of people, because of its more useful theoretical properties. In a series of papers P. Landin [1965], [1966] and [1966a] emphasised the use of the lambda calculus for describing the operational semantics of some imperative languages like ALGOL-60. Moreover he designed a reduction machine: the SECD-machine [1964]. In the meantime McCarthy had introduced the language LISP, having several aspects of functional languages, like e.g. recursion. However since LISP uses the quote and dynamic binding, it is not a purely functional language. Nevertheless the success of LISP made people aware of the usefulness of recursion in programming. Finally it was Backus [1978] who emphasised that functional languages allow the programmer to use a more flexible logical intuition and reasoning than is possible with the imperative languages.

Functional languages, by contrast to the imperative ones, use a semantic model that is alien to the behaviour of the physical components of a computer (processor, memory). Therefore these languages have to be realised at some level by interpretation on an imperative machine. This can be done in two ways. 1. By a software implementation operating on a traditional machine; 2. by specially built hardware that supports directly the functional semantics. Of the first kind we mention the implementation of the language SASL by D. Turner [1979] and the implementation of the languge LML by a group at Chalmers Technological University (the G-machine; see Augustsson [1984], Johnsson [1984]). An implementation on dedicated hardware is the commercial machine NORMA of Burroughs company, based on Turners implementation mentioned above.

Unlike the imperative model that is based on sequential state transitions, the reduction model that realises functional semantics is not inherently sequential. Therefore implementations on parallel hardware are theoretically possible. However, since there are considerable practical difficulties, no efficient parallel implementation has been realised yet (above examples are all on sequential

machines). Several research projects are on their way to develop parallel architectures for the evaluation of functional programs.

I.1.2 What Are Functional Programs?

Functional programs have as aim to compute the value of a function at some arguments. The arguments (and value) may be integers, characters or other symbolic expressions.

I.1.2.1 A functional program is an expression (representing the algorithm together with the input data) that is modified according to some given rules (reduced) step by step, until no more reductions are possible and the so called normal form (the output) is obtained.

The reduction rules come in two kinds: predefined rules and user-defined ones. Among the predefined rules are those that deal with the standard arithmetic operations.

I.1.2.2 Example. - The most natural way of evaluating an arithmetic expression will illustrate the process of reduction.
Let E = (13+1)*(13-1) + (5+3)*4. Then reducing E from right to left runs as follows. A single reduction step is denoted by the symbol '->'. The symbol '->>' denotes many (possibly zero) reduction steps.

```
(13+1)*(13-1) + (5+3)*4  ->  (13+1)*(13-1) + 8*4
                         ->  (13+1)*(13-1) + 32
                         ->> 14*12 + 32
                         ->  168 + 32
                         ->  200.
```

Note that this process describes the computation much more clearly than the usual imperative way of evaluating expressions, copying values to a stack or into registers. The above example is typical for dataflow computations. Functional programs in general contain more complicated expressions.

The output is independent of the order in which the reductions are performed (Church-Rosser property). Therefore the output is uniquely determined. For non-deterministic programs research is presently done on reduction systems that do not satisfy the Church-Rosser property.

I.1.2.3 Another Selfexplanatory Example -

```
The (2+2)-th member of (sort <3,5,2,1>) ->>
The  4-th    member of <1,2,3,5>        ->
5
```

For this program two subroutines are used: 'sort' and 'member of'. We see clearly the possibility for parallel evaluation.

I.1.2.4 In some functional languages the set of user defined rules is empty. In that case the class of predefined rules should be powerful enough to be able to represent functional programs.

In the informal functional programs of section I.2 there is a possibility for the programmer to define his own rules. These user defined rules are part of the functional program. In the languge TALE there will be only predefined rules. The expressions written by the user will encode their necessary reduction behaviour.

I.1.3 Qualities of Functional Programs

Functional programs have no side effects. This means that the execution of part a program does not interact with other parts via some global 'machine state'. Therefore the meaning of other parts of the program is preserved. In particular there are

no input/output statements,
no goto statements,
no assignments.

Since there are no side effects in functional programs, these are said to be referentially transparant: the meaning of a part of a program is independent of the evaluation of the rest of the program.

Because of the mentioned qualities the following two claims are made about functional programming.

(i) It is good for writing structured software; better than the imperative languages.

(ii) Programs in a functional language can in theory be evaluated in parallel (reducing several subexpressions simultaneously). With a proper implementation high performance might result.

As a motivation for claim (i) we mention the following. It is a well accepted programming methodology that program components should have simple and easily defined interfaces with each other. In functional programming the information entering into a program component is entirely contained in the values of its subexpressions. Similarly, the effect of the evaluation of such a component is contained only in the resulting value it yields. In other words, the interfaces are simply values, and a program part may be completely specified by the mapping of input values to result values that it realises. Though this does not force the interface to be small or simple, (values may have a complicated structure) it does force it to be very explicit. The methodological advantages of functional programming can be summarised by the term 'compositionality'.

As stated before, there is also a price to pay for these two good qualities: there are no input/output statements.

One way to deal with this is the so called standard solution The purely functional reduction machine is attached to a von Neumann host machine and an imperative environment deals with I/O and interactive programs.

Another solution is to build on top of the reduction machine special imperative features compatible with the reduction machine itself. An example is the SASL command INTERACTIVE, see Turner [1979]. Research is being done to find other solutions.

I.2 SIMPLE FUNCTIONAL PROGRAMS

I.2.1 Application

Functional languages are sometimes called applicative languages. If F and A are expressions, then

F A

is another expression: 'F applied to A'. Think of F as function and of A as the argument.

It was noted by Schönfinkel that functions of more arguments can be reduced to unary functions. Say we have a function g which has at its two arguments A, B the value g(A,B). Now we allow as expression

G A

that will be considered as the unary function G' with

G' B = g(A,B).

We then have

(G A) B = G' B = g(A,B).

In general we let the expression

H A1 ... An

denote

(..((H A1) A2) ... An)

(association to the left), which in view of the above represents an n-ary function applied to A1,...,An.

I.2.2 Recursion

As a comparison we represent the function factorial in FORTRAN and in a functional language.

FORTRAN:

```
      INTEGER FUNCTION FAC(X)
      INTEGER X
      FAC = 1
      DO 100 I=1, X
      FAC=FAC*I
  100 CONTINUE
      RETURN
      END
```

In functional programming we may write

```
(1)     fac n -> if zero n then 1
                 else n * (fac (n-1))
                 fi
```

and ask for the value of 'fac 3' for example. The reduction to the normal form (value) is then as follows.

```
fac 3 ->    if zero 3 then 1 else 3 * (fac (3-1)) fi
      ->    if false  then 1 else 3 * (fac (3-1)) fi
      ->    3 * (fac (3-1))
      ->    3 * (fac 2)
      ->    3 * [if zero 2 then 1 else 2 * (fac (2-1)) fi]
      ->    3 * [if false  then 1 else 2 * (fac (2-1)) fi]
      ->    3 * [2 * (fac (2-1))]
      ->    3 * [2 * (fac 1)]
      ->    3 * [2 * [if zero 1 then 1 else 1 * (fac (1-1)) fi]]
      ->    3 * [2 * [if false  then 1 else 1 * (fac (1-1)) fi]]
      ->    3 * [2 * [1 * (fac (1-1))]]
      ->    3 * [2 * [1 * (fac 0)]]
      ->    3 * [2 * [1 * [if zero 0 then 1 else 0 * (fac (0-1)) fi]]]
      ->    3 * [2 * [1 * [if true   then 1 else 0 * (fac (0-1)) fi]]]
      ->    3 * [2 * [1 * 1]]
      ->>   6.
```

Although this does not look appetising, we should realise that this is a complete trace of the evaluation of fac 3 on the level of machine code. Doing something similar for the FORTRAN program is worse, since it involves at each step a display of the status of every variable in the program, whether involved in this computation or not. The code for fac 3 is at least mathematically understandable.

For the above reduction path we need the following predefined reduction rules.

(2) if true then P else Q fi -> P
(3) if false then P else Q fi -> Q
(4) zero 0 -> true
(5) zero n -> false, for n not zero

(where n is the denotation for the integer n) and moreover

(6) 3 * 2 -> 6
(7) 3 - 1 -> 2

etcetera. These reduction rules can be written in a purely applicative way:

if B then P else Q fi

can be considered as a notation for

If B P Q

and

3 * 2 -> 6

is a notation for

* 3 2 -> 6

(Polish notation). The reduction rules (2)-(7) will be standard for a functional language. (Rule (6) for example is an instance of a rule scheme. That is, for every numbers n and m with representations n and m we have as a predefined rule

n * m -> n*m.

Although this are infinitely many rules, their action is computable.) On the other hand (1) is given by the programmer.

A simpler version of (1) is

```
fac 0   ->  1;
fac n   ->  n * fac (n-1) {for integer arguments >0}.
```

I.2.3 List Operations

I.2.3.1 Primitives for Lists - Inspired by LISP we introduce the following.

```
Primitives:     <>, Cons, Hd, Tl, Null.
Rules:          Hd (Cons a b) -> a;
                Tl (Cons a b) -> b;
                Null <> -> true;
                Null (Cons a b) -> false.

Abbreviations:  a:b          = Cons a b;
                <a>          = a:<>;
                <a1,...,an>  = a1:<a2,...,an>
                             = a1:(a2:...(an:<>).
```

Then

```
Hd <a1,...,an> ->> a1;
Tl <a1,...,an> ->> <a2,...,an>;
a:<b1,...,bn>  =   <a,b1,...,bn>.
```

Now we will write some selfexplanatory mini programs. These should be taken seriously, since functional programs compose nicely to larger programs. For some operations on the integers we assume that they are given as predefined rules.

I.2.3.2 A Simple Applicative Language. - We will write our functional programs using the following language. There are constants and variables that are composed to terms using application. A rule is of the form

F t1 ...tn -> s

where F is a constant and t1,...,tn,s are terms. A program consists of some rules together with a term to be reduced. In BNF the grammar is as follows. The notation used to describe the grammar should be selfexplanatory. If not see II.2.1.

```
<atom>          ::= <variable> | <constant> | "(",<term>,")".
<term>          ::= <atom> | <term>," ",<atom>.
<rule>          ::= <LHS>,"->",<term>.
<LHS>           ::= <constant> | <LHS>," ",<atom>.
<declaration>   ::= <rule> | <declaration>,";",<rule>.
<program>       ::= <term>,"?" | <declaration>,"end",<term>,"?".
```

Programs are subject to the following conditions. The constants and variables form disjoint sets of identifiers. The first constant in a rule is said to be defined by that rule. The constants in the term of a program should either be defined in a predefined rule or in a rule in the declaration. A term in a program may not contain any variables. There are some relatively simple conditions on a rule declaration that warrant that terms have unique normal forms (if any). However, since we will omit userdefined rules in TALE, we will not discuss these.

I.2.3.3 The Length of a List -

```
Aim:            size <a1,...,an> ->> n.
Program
                size <>     -> 0;
                size (a:b) -> 1 + (size b)
                end

                size <2,5,33,1>?
```

The value will be 4.

As with the function fac, these two rules can be combined into one. So we have the alternative declaration.

size x -> if Null x then 0 else 1 + (size (Tl x)) fi

Clearly the original definition with paternmatch is preferable.

The actual term to be reduced is of course irrelevant for our purposes. Therefore in the following mini programs we will give only the rule declaration. These will be denoted by 'Def'.

I.2.3.4 Concatenating Two Lists -

```
Aim:            <a1,...,an> ++ <b1,...,bn> ->> <a1,...,an,b1,...,bn>.

Def:            x ++ y = app x y;
                append <> c -> c;
                append (a:b) c -> a : (append b c).
```

I.2.3.5 Reversing a List -

```
Aim:            reverse <a1,...,an> ->> <an,...,a1>.

Def:            reverse <> -> <>;
```

```
                reverse (a:b) -> (reverse b) ++ <a>.
```

For a more efficient reverse, see exercise I.2.6.1.

I.2.3.6 Applying a Function to All Elements of a List -

```
Aim:            map f <a1,...,an> -> <f a1,...,f an>.

Def:            map f <>    -> <>;
                map f (a:b) -> (f a) : (map f b).
```

Note that the function f is an argument of map. The list <a1,...,an> is an argument to the application 'map f'. Here we make use of the convention in I.2.1.

I.2.3.7 The Sum of a List of Numbers -

```
Aim:            sum <a1,...,an> ->> a1 + ... + an.

Def:            sum <> -> 0;
                sum (a:b) -> a + (sum b).
```

I.2.3.8 The Product of a List of Numbers -

```
Aim:            prod <a1,...,an> ->> a1 * ... * an.

Def:            prod <> -> 1;
                prod (a:b) -> a * (prod b).
```

I.2.3.9 The function sum and product are obtained in a similar way. Therefore we would like to have a function foldr that specialises to these two.

```
Aim:            foldr plus 0 = sum; foldr times 1 = prod.

Def:            foldr f c <>    -> c;
                foldr f c (a:b) -> f a (foldr f c b);
                plus n m   ->  n + m;
                times n m  ->  n * m.
```

E.g. foldr f c <1,2,3> ->> f 1 (f 2 (f 3 c)).

I.2.4 Infinite Lists

Consider the sequence of squares

```
                0, 1, 4, 9, ....
```

Let sq satisfy for all numbers n

```
                sq n ->> n * n.
```

A way of coding the elements of the sequence as an infinite list is as follows. Let

```
                L f n  ->  (f n) : (L f (n+1))
                List f ->  L f 0,
```

then

```
                List sq ->  L sq 0
                        ->  sq 0 : L sq 1
                        ->  sq 0 : (sq 1 : L sq 2)
                        ->  sq 0 : (sq 1 : (sq 2 : L sq 3)
                        ........
                        ->  sq 0 : (sq 1 : (sq 2 : ... L sq n)..).
```

I.2.4.1 Definition. - Given a term F, the '<u>infinite list</u>' of F 0, F 1, ... notation

```
<F 0, F 1, ... >,
```

is the term 'List f', with

```
List f -> L f 0
where L f n -> f n : L f (n+1) end.
```

Although infinite lists do not have a normal form, they can be used within an expression having a normal form.

I.2.4.2 There is a term Sub such that

```
Sub i <F 0, F 1, ... >  ->> F i.
```

Proof.

```
Def:        Sub 0  l  ->>  Hd l;
            Sub n  l  ->>  Sub (n-1) (Tl l) {for integer arguments >0}. #X#
```

We see that expressions may have a normal form altough an infinite reduction is also possible. This implies that some care is needed when reducing expressions. This topic will be discussed in section I.4.3.3.

I.2.5 Sets

I.2.5.1 Definition - Lists, finite or infinite, can be used to represent sets. The set corresponding to <a0, a1, ... > is {a0, a1, ... }. That is, we disregard the order and repetitions.

I.2.5.2 Proposition - There is a term 'In' such that for a list l of objects for which there is a definable equality predicate (for example a list of integers or of characters) we have

```
In a l  ->>  true     if a is in the set corresponding to l;
        ->>  false    else.
```

Proof.

```
Def:        In a <>     ->  false;
            In a (b:c) ->  a=b  or  (In a c);
            where  p or q -> if p then true else q fi end. #X#
```

I.2.5.3 Proposition (separation) - Let the set A be given as a list. Let be a Boolean function on A (i.e. for n in A one has that P n reduces to true or to false). Then one can represent as a list the set

(1) {x in A | P x}.

In fact there is a term Sep such that

```
Sep A P
```

reduces to the list representing (1).

Proof.

```
Def:        Sep <> P      ->  <>;
            Sep (a:b) P   ->  if P a then a : (Sep b P)
                              else Sep b P fi. #X#
```

The second line can be written in a shorter way:

```
Sep (a:b) P  ->  let c = Sep b P in
                 if P a then a : c
                        else c
                 fi.
```

I.2.5.4 Proposition (replacement) - Given a set A and a function F. Then one can represent as a list the set

(2) { F a | a in A }.

In fact there is a term Rep such that

```
Rep A F
```

reduces to the list representing (2).

Proof.

```
Def:        Rep <> F      ->  <>;
            Rep (a:b) F   ->  F a : (Rep b F). #X#
```

Note that Rep is very similar to 'map' above, which is a consequence of representing sets as lists.

I.2.6 Exercises

I.2.6.1 A more efficient reverse is as follows.

```
rev a -> f a <>
where   f <> c   -> c;
        f (a:b) c -> f b (a:c)
end.
```

Count the number of reduction steps necessary to normalise reverse (1,2,3,4) and rev (1,2,3,4) respectively.

I.2.6.2 Inspired by the algorithm quicksort we define

```
QS <> -> <>;
QS (a:b) -> QS (S a b) ++ <a> ++ QS (L a b)
where   S a <> -> <>;
        S a (b:c) -> if b<=a then b : (S a c)
                             else S a c
                     fi;
        L a <> -> <>;
        L a (b:c) -> if b>a then b : (L a c)
                            else L a c
                     fi
end
```

Find the reduction to nf of QS <4,5,3,2,7>.

I.2.6.3 Let L=<<k1,a1>,..,<kn,an>> with k1,...,kn integers be a 'keyed list'. L is 'ordered' if k1<=k2<= ... <=kn.

(i) Write a functional program O that orders a keyed list.

(ii) Suppose L is ordered. Write a program In that inserts a [k,a] at the correct place in L.

I.2.6.4 Simulation of Loops - Write a functional program 'For' that satisfies e.g.
For <1,3> f c ->> f 3 (f 2 (f 1 c)).

I.2.6.5 Write a functional program 'sub i l' satisfying
sub i <a1, ... ,ak> ->> ai,
if i is an integer between 1 and k.

I.2.6.6 The 'towers of Hanoi' is the following problem. There are three poles a, b and c. There is a set of n flat discs of different sizes with a hole in the middle. These discs are on a given pole in decreasing order, the largest one down. The problem is to move the set of discs to one of the empty poles by moving a top disc from any pole to any other pole with the restriction that it is not allowed to lay a larger disc on top of a smaller one. Write a functional program H such that H n a b c gives as output the list of required moves to solve the problem for n, moving them from a to b. A move can be specified as pairs, e.g. <a,c> means that the top disc on a is moved to c. [Hint: use recursion. A two line program is possible.]

I.2.6.7 The following program makes use of infinite lists and of the set notation.

```
Def:        From n  ->  n : From (n + 1)
            Pr      ->  sieve (from 2)
            where   sieve (a:b)  ->  a : { x in b | divides x a }
                    divides x a   {Comment. Tests whether x divides a.}
            end
```

What is the normal form of
Sub 7 pr
where Sub is as in I.2.4.2.

I.2.6.8 Assume that L1 and L2 are ordered lists of integers. Construct an expression merge such that
merge L1 L2
reduces to the ordered union of the sets corresponding to L1 and L2.

I.2.6.9 The 'Hamming problem'. Make an infinite list L of the integers of the set {2^x * 3^y * 5^z | x,y,z >= 0}. [Hint. Using I.2.6.8 find a G such that L -> 1 : G L gives a solution.]

I.2.6.10 Write a program, using the set notation, that given a list L reduces to a list of lists consisting of all the permutations of L.

I.2.6.11 Let A be a finite set given as a list and let B x for x in A be a family of sets. Let F be a binary function and let P be a binary boolean predicate. Construct a set C functionally depending on A,B,F and P such that

C = { F x y | x in A and y in B x such that P x y }.

I.3 TYPES

I.3.1 Simple Types

Types are introduced to be assigned to expressions denoting functional programs. Each legal expression will get a unique type. The type of an expression is something like the dimension of a physical quantity (e.g. the dimension of speed is meter/second). In physics an equation can often be recognised as incorrect because the dimensions do not match. Similarly in programming types serve their role in helping to achieve correctness.

In this subsection we start explaining 'simple types', essentially a subset of the types that will be used eventually. This in order to make clear the idea. The simple types are sufficient for all the miniprograms in subsection I.2.3.

I.3.1.1 Simple types are defined as follows.

```
<simple type>    ::= <primitive type>|<list type>|<function type>.
<primitive type> ::= "int" | "bool".
<list type>      ::= "list",<type>.
<function type>  ::= "(",<type>,"->",<type>,")".
```

Notation. t,s,... denote arbitrary types; it is convenient to write

```
                t1 -> t2 -> ... -> tn -> s
```

for

```
                (t1 -> (t2 -> ... (tn -> s)..)).
```

This will be consistent with the notation

```
                F a1 ... an = (..((F a1) a2) ... an).
```

The type forming operator 'list' binds more strongly than '->'.

I.3.1.2 Examples of Types -

```
int;
int -> int;
bool -> int -> int;
list int;
list list int;
list bool -> int.
```

I.3.1.3 The intended meaning of the types is as follows. Each type t indicates a set D(t). The expressions having t as type will denote elements of this domain.

D(int)	is the set of integers {...,-2,-1,0,1,2,...};
D(bool)	is the set of truth values;
D(list t)	is the set of sequences of elements of D(t);
D(t -> s)	is the set of functions from D(t) to D(s).

I.3.1.4 Type Assignment - To each subexpression A of a legal functional program we want to assign a type, to be denoted by type(A). This type assignment should satisfy the following conditions.

- type(true) = type(false) = bool;
- int = type(0) = type(1) = type(-1) = ...;
- type(<E1,E2,...>) = list type(Ei), for all i;
- if an application F A occurs in the program, then type(F) = type(A) -> type(F A);
- both sides of a reduction rule should have the same types.

I.3.1.5 Now we will give examples how simple types can be assigned to some of the programs in section I.2.3.

(i) Consider the rule 'zero 0 -> true'. The conditions in I.3.1.4 imply that
type(zero 0) = type(true) = bool,
type(0) = int;
Hence
type(zero) = int -> bool.
Note that any other rule for 'zero', i.e. 'zero 1 -> false' is consistent with this type assignment.

(ii) type(plus) = int -> int -> int. Indeed, since 'plus 1 1 -> 2' is a reduction rule, we have
type(plus 1) = type(1) -> type(2) = int -> int,
type(plus) = int -> (int -> int) = int -> int -> int.

(iii) The expression 'size' finds the length of a list. If it is a list of integers, then type(size) = list int -> int. However the same expression 'size' works on all possible lists and we would like to write
type(size) = list t -> int
for arbitrary t. For the moment being we allow this ambiguity and say that type(size) is parametrised (by another type). In the next subsection we come back to this topic.

(iv) Write 'If B P Q' for 'if B then P else Q fi'. Since 'If true P Q -> P' and 'If false P Q -> Q', we must have type(P) = type(Q) = t, say. Then
type(If) = bool -> t -> t -> t.

(v) Now the reader can easily check the following type assignments.
type(fac) = int -> int;
type(append) = list t -> list t -> list t;
type(map) = (s -> t) -> list s -> list t.

I.3.2 Polymorphic Types

In the previous subsection we wanted to write
type(size) = list t -> int
for all types t. This is in conflict with our requirement that each expression has a unique type. In the second order typed lambda calculus, due to Girard and Reynolds, this problem is solved by giving expressions like 'size' a so called polymorphic type
@a list a -> int
that can be specialised to an arbitrary type t. This specialisation will be denoted by 'size$t'

I.3.2.1 The simple types are extended to types as follows.

```
<type> ::= <type variable>|<primitive type>|<list type>|
           <function type>|<polymorphic type>.
<type variable>    :== <bold identifier>.
<primitive type>   ::= "int" | "bool".
<list type>        ::= "list",<type>.
<function type>    ::= "(",<type>,"->",<type>,")".
<polymorphic type> ::= "@",<type variable>,<type>.
```

Notation. a,b denote arbitrary type variables.

I.3.2.2 The intended interpretation of types is extended as follows.

$$D(@a\ t(a)) = X_a\ D(t(a)),$$

i.e. the generalised cartesian product, whose elements are maps, giving for any (type) a an element of D(t(a)). Sometimes @a t(a) is called the 'universal

quantification' over types.

I.3.2.3 The formation rules for expressions are extended as follows.

If type(E) = @a t(a), and s is a type, then E$s is an expression of type t(s). Here t(s) denotes of course the substitution of s for a in t(a).

The program for 'size' now has to be given as follows.

```
type(size) = @a list a -> int
size$a <> -> 0
size$a (x:y) -> 1 + (size$a y)
```

and these rules may now be consistently typed by type(x)=a, type(y)=list a. A program for size that works only for lists of integers is as follows.

```
type(size') = list int -> int
size' <> -> 0
size' (x:y) -> 1 + (size' y).
```

Now of course we have type(x)=int, type(y)=list int.

I.3.2.4 Now we can give unique types to the expressions in the miniprograms of subsection I.2.3. E.g.

```
type(If)   = @a bool -> a -> a -> a;
type(Cons) = @a list a -> list a -> list a;
type(Hd)   = @a list a -> a;
type(Tl)   = @a list a -> list a.
type(map)  = @a@b ((a->b)-> list a->list b)
```

I.3.2.5 If we want to use Cons for integers we officially have to write

```
Cons$int 3 <2,7> -> <3,2,7>.
```

However, since the context is such that we can derive the necessary specialisation of Cons to Cons$int, it will be allowed that the '$int' be omitted. In the description of TALE it will be stated exactly under which circumstances this is possible.

I.3.3 Data Types

Now we extend the set of types by adding new ways of type formation: tuples, unions and arrays.

```
<type> ::= <type variable>|<primitive type>|<list type>|
           <function type>|<polymorphic type>|
           <tuple type>|<union type>|<array type>.
```

I.3.3.1 The syntax of the new types is informally described as follows. A tuple type is of the form

(t1,...,tn)

where the ti are types and n>=2 or n=0. A union type is of the form

(t1|...|tn)

where the ti are types but now n>=2. An array type is of the form

[,...,]t

where t is a type and there are n>=1 many blanks. This type is sometimes informally denoted by [n]t.

I.3.3.2 Tuple Types - The intended interpretation of these types is given by

D((t1,...,tn)) = D(t1) X ... X D(tn),

where X denotes the (ordinary) cartesian product.

The formation of expressions is now extended as follows. If E1,...,En are expressions of type t1,...,tn respectively, then

(E1,...,En)

is an expression (a tuple) of type (t1,...,tn).

The reader may expect some primitive projection functions like

P3 (x1,x2,x3) -> x3.

However, the use of these these will be avoided by allowing in the defining reduction rules for some functions the use of tuples of variables as arguments. E.g.

Discr (a,b,c) -> b^2 - 4*a*c

is a correct defining reduction rule. Such definitions may also make use of nested tuples of varaibles. E.g.

Inprod ((a1,a2),(b1,b2)) -> (a1*b1)+(a2*b2).

Discr as declared above is sometimes called a 'polyadic' function having type (int,int,int) -> int. We still may declare a

Discr' a b c -> b^2 - 4*a*c

having type int -> int -> int -> int. Pattern match inside such declarations is not legal. E.g.

G (a,b,(0,c)) -> a*b + c

is not allowed. Pattern match will be possible using unions.

The systematic name for the type of the empty tuple () would be (). In order not to confuse this type with its unique element (), we will write * for the type of ().

I.3.3.3 Union Types - The intended interpretation of these types is given by

D((t1|...|tn)) = D(t1) + ... + D(tn),

where + denotes the disjoint union. This operation is defined by

A + B = {(1,x)| x in A} U {(2,x)| x in B}.

That is, disjoint copies of A and B are made and these are united.

For a disjoint union of type (t1|t2|t3), say, there are canonical embeddings F1, F2, F3 with

type(Fi) = ti -> (t1|t2|t3).

The syntax for expressions will be extended to include these. The image of an E of type t3 under F3 will be denoted by

(t1|t2|E)

and these kind of expressions will be allowed in our functional programs. It is a bit anoying fact of life that it is necessary to specify the types of the variants that are not taken. The embedding function F3 itself can be defined by

type(F3) = t3 -> (t1|t2|t3);
F3 x -> (t1|t2|x).

An expression like (t1|t2|E) may be abbreviated to (| |E), if the types t1 and t2 can be deduced from the context.

A function G on a union type can be specified by its working on the various components.

Let E be of the union type (t1|t2|t3) and Gi of type ti -> s. Then

case E of G1|G2|G3 esac

is a legal expression of type (t1|t2|t3) -> s having as reduction rules

```
case (E1|t2|t3) of G1|G2|G3 esac -> G1 E1;
case (t1|E2|t3) of G1|G2|G3 esac -> G2 E2;
case (t1|t2|E3) of G1|G2|G3 esac -> G3 E3.
```

This mechanism will be used for expressing pattern match in TALE.

As an example of the use of the union type we will construct a type for 'mixed' lists of objects that are either an integer or a list of integers. Define

```
t = list (int | list int).
```

We expect that an expression like

```
<2, 3, <5,1>,7, ...>
```

is of type t. This is not quite so. Let F1 and F2 be the canonical embeddings of type int -> t, list int -> t respectively. Then

```
<F1 2, F1 3, F2 <5,1>, F1 7, ...>
```

is the expression that we wanted. It will be convenient to give F1 and F2 mnemonic names:

```
F1 = this_is_an_int
F2 = this_is_a_list
```

Then our expression becomes

```
<this_is_an_int 2, this_is_an_int 3, this_is_a_list <5,7>,
 this_is_an_int 7, ...>.
```

The embeddings this_is_an_int and this_is_a_list are called constructors for the type (int | list int)

A map H of type t -> list int that squares the integers and take the sums of the lists in a mixed list can be defined as follows:

```
H l ->  map f l
where   f x ->  case x of sq | sum esac end
```

or using pattern match:

```
H l ->  map f l
where   f (this_is_an_int x) -> sq x;
        f (this_is_a_list l) -> sum l
end.
```

In this example we produce a list of integers from a mixed list. If we would want to make a new mixed list, say by squaring the integers and reversing the lists, then we would have to apply the embedding functions again. So a map H' of type t -> t that squares the integers and reverses the lists in a mixed list can be defined as follows:

```
H' l -> map f l
where   f x ->  case x of this_is_an_int.sq
                        | this_is_a_list.reverse esac end
```

(where '.' denotes function composition).
Using pattern match this can be written:

```
H' l -> map f l
where   H (this_is_an_int x) -> this_is_an_int(sq x);
        H (this_is_a_list l) -> this_is_a_list(reverse l)
end.
```

A nice example of both the use of tuples and unions is the function that computes the roots of a quadratic polynomial by the well-known formula. For the result there are two distinguished cases: either two real roots, or two complex roots. Now complex numbers are declared in the initial environment of TALE, so that we can use the type compl and the dyadic operator i building a complex number from its real and imaginary part: a i b gives the complex number a + i*b. The two distinguished cases for the result are encoded by a union type with two variants. In the first variant the result is a pair of reals, in the second a pair of complexes. So the result type is ((real,real)|(compl,compl)). This illustrates a general programming methodology to use unions whenever a result can take several distinguished variants, and to use tuples to deliver multiple results.

```
type(roots) =
        (real,real,real) -> ((real,real)|(compl,compl));
roots (a,b,c) ->
        let d= b^2 - 4*a*c in
        if d>=0
        then ( ( (-b+(sqrt d))/2*a , (-b-(sqrt d))/2*a )
             | (compl,compl)
             )
        else ( (real,real)
             | ( (-b/2*a)i(( sqrt(-d))/2*a)
               , (-b/2*a)i((-sqrt(-d))/2*a)
               )
             )
        fi
```

I.3.3.4 Array Types - Although lists are convenient data types, there is no random access to the elements of a list. Arrays are introduced in order to create the possibility of an efficient implementation. Whether this possibility can actually be realized in the context of a functional language is a topic of active research.

I.3.3.5 An n-block is an expression of the form

D = ((l1,u1), ..., (ln,un))

with l1,u1, ... all integer denotations. The intended meaning of D is the cartesian product

I1 x ... x In

with Ii = {k| li <= k <= ui}. That is, n-blocks correspond to rectangular subsets of int^n.

A tuple (k~) = (k1, ..., kn) is within D if k~ belongs to the interpretation of D.

I.3.3.6 Let t be a type. An array type is of the form [,...,]t. For example []t, and [,,]t. These types may be written also as [1]t and [3]t. An array A of type [n]t consists of n-block D, the descriptor of A, together with a set of relatively simple (to be explained later) expressions of type t indexed by the elements of D. Such an A is also called an n(-dimensional)-array.

For example, the matrix A0:

```
| 1 2 3 |
|       |
| 4 5 6 |
```

can be seen as [2]int array with descriptor D0 = ((1,2),(1,3)).

Of course the notation for A0 was informal. In fact we have no expressions denoting arrays, only ones that reduce to an array.

I.3.3.7 Subscriptions - If A is an [n]t array with descriptor D and (k˜) = (k1, ..., kn) is an n-tuple within D, then

A[k˜] = A[k1, ..., kn]

denotes the element of A with index k˜. For example

A0[2,1] = 4

the notation is extended to

A[k˜ ext F]

which reduces to A[k˜] if k˜ is within D, otherwise to F (k˜).

I.3.3.8 Tabulation - If D is an n-block and F : int^n -> t, then there is an obvious [n]t array A with descriptor D and with values at index k˜ the value of F k˜. This array will be obtained reducing the expression

tab D : F bat

For example, if F0 (n,m) = n*m, then

tab D0 : F0 bat

reduces to the matrix

| 1 2 3 |
| |
| 2 4 6 |

I.3.3.9 Another important operator on arrays is given by the expression

for' A : F 'rof.

It reduces to a new array B with the same dimension and descriptor as A and

B[k˜] = F(A[k˜])

In another version of 'for' F makes also use of the indices of the elements it operates on:

for A : F rof

reduces to C with

C[k˜] = F(k˜, A[k˜]).

The legal for expressions may be somewhat more complicated and will be discussed in the next section. Also several other array operations will be introduced there.

I.3.4 Recursive Types

We have seen that mixed lists consisting of integers or lists of integers can be modelled on the type list (int | list int). Objects of type "hereditary lists of integers" e.g.

L = <2, <1>, <3, <5,6> >, ...>

can not yet be modelled within our type system, since the nestings may grow arbitrarily deep. The recursive types will allow us to assign a type also to an expression like L.

I.3.4.1 If t = t(a) is a type (in which the type variable a occurs), then we like to have a 'recursive type' r, such that r = t(r). Such an r will be introduced by allowing

r = rectype a: t(a).

as a type formation rule. Moreover all equalities following from r = t(r) are postulated to hold.

I.3.4.2 Example - We want to introduce a type representing trees of integers. Let

T = rectype a: (int | (a,a)).

the this type satisfies

T = (int | (T | T)).

An object of this type essentially looks like

((3, (4,1)), 2)

and can be drawn as a tree

However, such an expression is not officially of type T, since not 3 but the canonical embedding of 3 in T has to be there. Again we choose mnemonic names for the constructors

```
leaf : int -> T = (int |(T,T))
node : (T, T) -> T
```

then the example becomes

```
node (node (leaf 3, node (leaf 4, leaf 1)), leaf 2)
```

I.3.4.3 Similarly we can construct list int as a recursive type:

```
list int= rectype ls: (* | (int, ls)),
```

where * is the type for the (unique) 0-tuple (). We write for the canonical embeddings

```
nil  : * -> list int
cons : (int, list int) -> list int
```

and abbreviate nil () as nil or as <>. Then

```
cons (7, nil)
cons (3, cons(7, nil))
```

are of type list int, as we were used to.

I.3.4.4 List as a Type Generator - More generally we may put

```
list $ t = rectype ls: (* |(t, ls)).
```

Then 'list' will be called a type generator and the types 'list $ t' will play the role of the primitive 'list t'. Therefore recursive types make the notion of list type superfluous.

Note that we cannot write

```
type(list) = @ a (rectype ls: (* |(a, ls))).
```

The map t -> list $ t is not definable. This by contrast to the map

```
t -> size $ t
```

that is definable and of type @ a (list a -> int). The expression list is called a 'type generator' that can only occur in the context list $ t, exactly as we were used to.

I.3.4.5 Now we can introduce the type for hereditary lists of type int, say:

```
hlist = rectype h: list $ (int|h)
```

For the constructors we choose the following names

```
is_int   : int -> (int | hlist)
is_hlist : hlist -> (int | hlist).
```

As an expression of type hlist we have

```
cons (is_hlist (cons (is_int 4, nil)
     ,cons (is_int 3, nil)
     )
```

This is the official version of < <4>, 3 >.

I.4 TOWARDS TALE

The language TALE consists of a core language and extensions. Programs written in the extensions can all be translated to the core language. Nevertheless these extensions are very useful, since they contain often used constructs. In fact most of the sample programs above are written in the extensions.

I.4.1 The Core Language

I.4.1.1 Abstraction - In subsection I.1.2.4 we mentioned that TALE has a fixed predefined reduction system. Nevertheless it is able to represent the functional programs like those in I.1. This is achieved by using lambda abstraction. The greek letter lambda will be printed as `.

Lambda abstraction works as follows. Consider the definition

```
square = `x -> x*x
```

(also written as square = `x.x*x) and the general rule of beta reduction

```
(`x.A)B    ->    A[x:=B]
```

where A[x:=B] denotes the result of substituting B for x in A.
Hence `x.A or `x -> A intuitively denotes the function that maps x to A. Therefore indeed `x -> x * x acts as the function square. A particular instance of the beta reduction rule is

```
(`x->x * x) 3 -> 3 * 3.
```

With typing this last reduction will be written as

```
(`int x -> x * x) 3 -> 3 * 3
```

to indicate that x is of type int.

A function like

```
p (a,b) -> a + 2 * b
```

can be obtained by setting

```
p = `(a,b) -> a + 2 * b
```

(or with the types p = `(int, int) (a,b) -> a + 2 * b)
If we extend the beta reduction rule to obtain reductions like

```
(`(a,b) -> a + 2 * b) (3,4) -> 3 + 2 * 4
```

If on the other hand we declare

```
p' a b -> a + 2 * b,
```

i.e. type (p') = int -> int -> int, then we can take

```
p' = `a.(`b. a + 2 * b),
```

indeed,

```
p' 3 -> `b. 3 + 2 * b
```

hence

```
p' 3 4 -> 3 + 2 * 4
```

I.4.1.2 Recursion - A recursive function declaration like

```
fac x -> if zero x then 1 else x * fac (x - 1)) fi
```

also has to be captured in TALE. We cannot write as definition

```
fac = `x -> if zero x then 1 else x * (fac (x -1)) fi
```

since fac occurs also in the right hand side of this equation. We introduce in the core language a new abstractor 'rec f:' with the reduction rule

```
rec f:E -> E[f := rec f:E].
```

Then we can put

```
fac = rec f: `x -> if zero x then 1 else x * (f(x - 1))fi
```

This gives the disired effect:

```
fac 3 -> (`x -> if zero x then 1
                          else x * (fac (x - 1))fi)3
      . ...
```

Another example of recursion is

```
ONES = rec f: (1:f)
```

this gives an infinite list of 1's:

```
ONES -> (1:ONES) -> (1:(1:ONES)) -> (1:(1:(1:ONES)))...
```

Mutual recursion can be dealt with in TALE, as will be explained later.

I.4.1.3 Polymorphism - Let I be the polymorphic identity function of type @t t -> t. I satisfies

```
I$t = `t x -> x.
```

We introduce in TALE the function I as follows

```
I = %t `t x -> x.
```

Here %t is a abstractor with t a type variable that operates on expressions. We have

```
type (%t E) = @t type(E)
```

and the reduction rule

```
(%t E) $ s -> E[t := s]
```

Another example, combining the three kinds of abstraction:

```
size = %t (rec f : `(list$t)x -> case x
                                 of `x -> 0
                                 |  `x -> 1 + (f x)
                                 esac
)
```

I.4.1.4 Other Features of the Core Language -

- Primitive types. There are more primitive types than we have discussed so far. In TALE we have

```
<primitive type> ::= "int" | "char" | "real" | <type variable>
```

The type 'char' is for expressions that are character denotations. The denotation of a, b, ... is 'a,'b, ... The type 'real' is for expressions that denote real numbers in some floating point notation. The type 'bool' is not present, since it can be defined.

- Error. There is an expression 'error' of type @t t. It will be used in reductions like

```
div (1,0) -> error "divide by 0"
```

- Comments. Text within curly brackets {} can be used as a comment on the program.

- Case-in expression. The format is

```
case E in F0, F1, ..., Fn out G esac
```

and has as reduction rule

```
case i in F0, F1, ..., Fn out G esac -> Fi  (if 0 <= i <= n)
                                     -> G i (else)
```

where i is the denotation of some integer i.

Using this we can define the test for zero:

```
zero = `x -> case x in true out `y -> false esac ;
```

or the Fibbonaci sequence

```
fib = rec f: `int x . case x in 1,1
                        out `y. f (y - 1) + (f (y -2)) esac.
```

The last expression fib may also be declared as follows (using an extension)

```
fib 0 = 1
  | 1 = 1
  | n = fib (n - 1) + (fib ( n - 2))
```

I.4.1.5 Array Operations - There are four categories of operators on arrays that we still have to discuss:

1. The operation by the 'for' expression;
2. The descriptor operations;
3. The update and exchange;
4. Operations on one dimensional arrays.

1. The 'for' expressions as introduced is the last section can be made more powerful. Remember that for A : F rof reduces to an array A' with the same descriptor as A and

$$A'[i^{\sim}] = F((i^{\sim}), A[i^{\sim}]).$$

Now define the following two operations on arrays. First let A1, A2 both have the same descriptor D. Then A1 ++ A2 is the array with the descriptor D and satisfying

$$(A1 \mathbin{++} A2)\ [i^{\sim}] = (A1[i^{\sim}], A2[i^{\sim}]).$$

Secondly let A, B be arrays with descriptor

((l1,u1), ...) and ((l'1,u'1), ...)

respectively. Then A**B is the array with descriptor

((l1,u1), ..., (l'1,u'1). ...)

and satisfying

$$(A**B)\ [i^{\sim}, j^{\sim}] = (A\ [i^{\sim}], B[j^{\sim}]).$$

Now a typical form of the strengthened for expression looks like

for A1||A2, B1||B2||B3, C1||C2 : F rof

and reduces to the same array as

for ((A1 ++ A2) ** (B1 ++ B2 ++ B3) ** (C1 ++ C2)) : F rof

in particular

for A1||A2, B1||B2||B3, C1||C2 :F rof [i~, j~, k~] ->
F((i~, j~, k~), ((A1[i~], A2[i~]), (B1[j~], B2[j~], B3[j~]), (C1[k~], C2[k~]))).

We have chosen to define 'for' with this strength rather than building it up from the simple 'for' using the operations ++ and **, because it is more efficient to form directly the intended result rather than first the intermediate arrays like A1++A2 etcetera.

2. The descriptor transformations form a class of operations on arrays that do not change the components, but change the dependency of those components on the indices. The dependencies may be permuted, cut down (taking a subblock of the one described by the descriptor) or rearranged (e.g. considering a given matrix as a row of rows). The precise definition and format of these operations are to be found in subsection II.2.6.3.

3. The update and exchange operations on a array A have the following formats.

Update	A([i~] := a)
Exchange	A([i~] <-> [j~])

Their meanings are explained in the following examples.

```
| 1 2 3 |                          | 1 2 3 |
|       | ([2,2] := 7) ->          |       |
| 4 5 6 |                          | 4 7 6 |

| 1 2 3 |                          | 1 5 3 |
|       | ([2,2] <-> [1,2]) ->     |       |
| 4 5 6 |                          | 4 2 6 |
```

The exchange operators are extended to make exchange of sub-arrays possible.

It is sometimes felt that these two operations are not functional. That is however incorrect. An update is as functional as is the reduction

```
square (-3) -> 9
```

where also the -3 is lost. It is on the other hand true that these operations have their price. If an array is shared (see section I.4.3) then one must make a copy for the update. How high this price is has to be seen. In many programs the arrays do not need to be shared. However we have to admit that it might be expensive to detect whether an array is shared.

4. A one dimensional array is also called a row. One operation useful for rows is their explicit description; the array display:

```
[[a1, ..., an]]
```

denotes the array with descriptor ((1,n)) and as component ai at index i.

Other useful operations on rows are concatenation, splitting and fold (similar to fold for lists). There are some other ones as described in subsection II.2.6.5.

I.4.2 Extensions of the Core Language

The core language of TALE has several extensions to make programming more flexible. Some of those we have encountered already. All the constructs in the extension can be translated to the core language. This section describes informally the essential parts of the extensions, following the order of section II.3.

I.4.2.1 Miscellaneous Extensions

- Booleans and conditional. We define the type

```
bool = (*|*).
```

That is, the disjoint union of the singleton set {()} and itself giving indeed a two element set. We set

```
true  = (()|  )
false = (  |())
```

The canonical elements of type bool. Using this we can define

```
if b then x else y fi =

case b of `z.x | `z.y esac
```

Then indeed we have e.g.

```
if true then x else y fi ->
case (()|  ) of `z.x | `z.y esac ->
(`z.x)() ->
x.
```

- Strings. These are represented as a row of characters. E.g. we define

```
"cat" = [[ 'c, 'a, 't]]
```

- Array subscription. The official subscription operator is A[i~ ext F] giving F i~ if i~ is not within the description of A. The simple version A[i~] is in fact

defined by

A[i~] = A[i~ ext '(x~) -> error]

which because of some later extensions may be written as

= A[i~ ext - -> error] .

- Descriptor transformations. Notations are introduced for applying several of these consecutively. E.g. <[2,1]> transposes a matrix and <[][]> considers it as a row of rows. The combined action performed by <[2][1]> considers the matrix as a row of columns.

I.4.2.2 Simple Declarations

- let and where. If we need a locally declared identifier within some expression we may write

```
let d = b^2 - 4*a*c in (- b + (sqrt d)) / 2*a
```

or

```
(- b + (sqrt d)) / 2*a where d = b^2 - 4*a*c end
```

Both expressions are abbreviations of

```
('d.(- b + (sqrt d)) / 2*a) (b^2 - 4*a*c) .
```

In a let expression several constants may be defined.

```
let  d = E1,
     e = E2
in E
```

stands for

```
let (d,e) = (E1, E2) in E
```

and this for

```
('(d,e) . E) (E1, E2).
```

If a constant declared by let depends on a previous one, then we may write

```
let  d = E1;
     e = E2
in E
```

standing for

```
let  d = E1 in let e = E2 in E .
```

I.4.2.3 Mutual Recursion and Recursive Declarations

Sometimes we want to introduce two functions that mutually depend recursively on each other. For example E and O, the characteristic functions on the even and odd numbers, can be defined by

```
E 0 = true
| n = O (n - 1)
O 0 = false
| n = E(n - 1)
```

This follows if we have

```
E = 'x . case x in true  out 'n -> O (n - 1) esac
O = 'x . case x in false out 'n -> E (n - 1) esac
```

Or in a more abstract way

```
E = F(E,O)
O = G(E,O).
```

Such mutually recursive equations may be solved by putting

```
(E,O) = rec(E,O) : (F(E,O), G(E,O))
```

where

```
rec(E,O) : (F(E,O), G(E,O))
```

stands for the espression Z equal to

```
rec z : (F(x,y), G(x,y))
    where x = (' (p0,p2) -> p1)z,
          y = (' (p1,p2) -> p2)z
    end
```

Indeed, if we abbreviate ('(p1,p2) -> pi)Z as Zi then by the property of rec z we

have

Z ->> (F(Z1,Z2) , G(Z1,Z2)).

Hence

Z1 -> F(Z1,Z2)
Z2 -> G(Z1,Z2)

so

Z = (Z1,Z2).

Now we can allow recursive declarations:

let rec d = E1(d,e),
e = E2(d,e)
in E

is an extension that stands for

let (d,e) = rec (d,e) : (E1(d,e), E2(d,e)) in E

I.4.2.4 Type Declarations

If we want to define a new type, like we did with bool, then we may write

type bool = (*|*) in E.

This will be translated to E with all occurrences of bool replaced by (*|*). Another declaration is for type generators, i.e. types depending on types. We may write

type list = %a rectype ls: (*|(a,ls)) in E.

This translates to E with every occurence of list$t within E, with varying t, replaced by the appropriate rectype ls: (*|(t,ls)).

I.4.2.5 Forgetful Extensions

There are extensions that introduce shorthand notations.

- `x(y,z).E stands for `x.(`(y,z) . E)
- `-.E stands for `x.E with x not occuring in E.
- In some cases we may omit the abstractor ` completely.

  case E in F1 , F2 out x -> x*x esac

  stands for

  case E in F1 , F2 out `x -> x*x esac
- Expressions like

  `x.case x of F1 | F2 esac

  may be condensed to

  case of F1 | F2 esac

  if x is not used in F1 nor in F2.
  Similarly case in F1 , F2 out E esac stands for `x.case x in F1 , F2 out E esac.
- Types may be omitted if we can deduce from the context the type of an expression. (For this reason the attributes strong and weak are introduced in the description of TALE). The same holds for the type specialisations of a polymorphic function:

  size $ int x . ...

  may be abbreviated to

  size x . ...

  if size has type @a(list$a->int), but we can deduce that x is of type list$int.

I.4.2.6 Constructors

- The use of this extension we already have encountered implicitly. It allows to give mnemonic names to the canonical embeddings in a union type. These embeddings will be called constructors. We write

constructors this_is_an_int, this_is_a_pair for (int | (int, int)) in E

This expression stands for E in which each occurence of this_is_an_int is replaced by `x.(x| ) and each this_is_a_pair by `z.(|z), i.e. by the canonical embeddings.

- If the two constructors this_is_an_int and this_is_a_pair are introduced as above we may write

```
case E of (this_is_a_pair(a,b)) -> a+b |
          (this_is_an_int x)    -> square x esac .
```

this then denotes

```
case E of `x -> square x | `(a,b) -> a+b esac.
```

The order of the constructors in their declaration determines the order of the so called case limbs.

- If one of the types in a union type is *, i.e. the singleton type with unique element (), then the constructor for that type does not stand for the embedding map f of type (* -> (* |...)), but rather for the value f () = (() |...) of type (* |...). So for example

```
constructors true, false for (* | *) in E
```

replace each occurence of true in E by (() |) and false by (| ()).

I.4.2.7 Heuristic Application and Pattern Match

- Heuristic application. Rather than writing

```
sq = `x -> x*x
```

to declare a function, we may write

```
sq x = x*x
```

Similarly

```
id $ a = `a x -> x
```

stands for

```
id = % a `a x -> x.
```

- Pattern match. Let t be the type (int | (int, int)) with constructors this_is_an_int and this_is_a_pair. We can declare a function f of type (t -> int) by writing

```
f (this_is_an_int x)      = square x
| (this_is_a_pair (a,b)) = a+b
```

This will be translated to

```
f = `x.case x of `x -> square x | `(a,b) -> a+b esac.
```

Similarly a pattern match for numbers

```
f 0 = 2
| n = square (n - 1)
```

translates to

```
f = `x.case x in 2 out `n -> square(n - 1) esac.
```

I.4.2.8 Lists

We have already seen that how lists are definable using recursive types. We put

```
type list = %a rectype lt: (*|(a, lt)) in
constructor nil, cons for list in E.
```

This will be translated such that

```
nil $ t and cons $ t
```

are the constructors for list $ t (We did not yet explain this feature of polymorphism within the constructors).

Moreover, the usual notational conventions are adopted for lists in TALE, except that as list brackets we use (< >).

I.4.2.9 Formulae

Rather than always writing

```
plus (a,b)  or  times (a,b)
```

we like to use the formulas

```
a + b     or    a * b
```

and declare priority rules indicating that

```
a + b * c = a + (b * c)
```

A mechanism for creating new operators (monadic and dyadic ones) together with their priority rules is introduced in an extension. For any identifier that denotes a function we may use an operator symbol instead (replacing function applications by formulae) if this is convenient. So we may declare operators globally or locally, or even use them to denote a function-parameter (e.g. in the definition of fold).

I.4.2.10 Abstract Types

Suppose we have introduced a type t together with some operations on it. Often we want to hide the actual details of the definition of t and allow only operations on the type t that are built up from the given functions. We can, do this by declaring a so called 'abstract type'.

We give an example. Suppose we want to introduce a constructed type for the natural numbers with zero, successor, predecessor and the test for zero. Now we could declare:

```
type nat = rectype a: ( * | a ) in
constructors zero, succ for nat in
let  pred = case nat of (zero) -> zero | (succ x) -> x esac
, is_zero = case nat of (zero) -> true | (succ -) -> false esac
in E
```

If we proceed like this, than in E we will have nat = (* | nat), and we may use the syntactical forms that this implies (e.g. the case of construct and the pattern matching). Now we may like to hide this concrete representation for nat, making it abstract, and accessible only by use of zero, succ, pred and is_zero. This can be achieved by writing instead:

```
abstype nat
    with nat zero,
         (nat->nat) succ, (nat->nat) pred,
         (nat->bool) is_zero
= rectype a: ( * | a ) {our implementation type}
    with type n = rectype a: ( * | a ) in
         constructors z, s for n in
         let p  = case n of (z) -> z | (s x) -> x esac
         , is_z = case n of (z) -> true | (s -) -> false esac
         in (z, s, p, is_z) {implementations for zero, succ, etc.}
in E
```

Now we have in E, that nat is an abstract type, showing no more representation that e.g. int or real. In E zero, succ, pred and is_zero are accessible, and have types expressed with nat as declared following the first with. The identifiers z, s, p, is_z as well as n, and their properties, are no longer visible.

This 'abstract declaration' is translated into the core language as follows:

```
(%nat `(nat zero,
         (nat->nat) succ, (nat->nat) pred,
         (nat->bool) is_zero
       ).E)
$ rectype a: ( * | a ) {our implementation type}
       (type n = rectype a: ( * | a ) in
        constructors z, s for n in
        let p  = case n of (z) -> z | (s x) -> x esac
        , is_z = case n of (z) -> true | (s -) -> false esac
        in (z, s, p, is_z) {implementations for zero, succ, etc.}
       )
```

Now, inside E we have that nat is a type variable and zero, succ, pred and is_zero are variables with types expressed using nat. Therefore the context E cannot make

use of the specific properties of n that we wanted to hide.

I.4.3 Reduction and Semantics

I.4.3.1 Reduction

A reduction relation on a set of expressions is generated by some 'contraction rules'. A contraction rule specifies for what expressions E1, E2 one has as an axiom

```
E1 -> E2
```

If E1 is part of a larger expression E(E1), and E1 -> E2, then E(E1) -> E(E2) is called a one step reduction. So

```
(`int x . x * x) 3 -> 3 * 3
```

is a one step reduction that is given by a contraction rule, but

```
2 + ((`int x . x * x) 3) -> 2 + (3 * 3)
```

is just a one step reduction. Finally the reduction relation ->> is the reflexive transitive closure of one step reduction. That is E ->> E' if and only if for some sequence E0, E1, ..., En one has

```
E = E0 -> E1 -> ... -> En = E'.
```

An expression E is in normal form is it has no reducible part. In TALE an expression is called root-reduced if it is either 'error', a denotation of an integer real or character, a tuple, union or array or a function (i.e. of the form ` ... or % ... or a predefined one). An expression is reduced if it is root-reduced and moreover for a tuple and a union the components are also reduced. In TALE only arrays are allowed with reduced components. This implies that reducing tab D : F bat all the F i~ have to be evaluated to reduced form. Some examples.

```
case 0 in 1 out `x . x esac      is not root-reduced;
(`x . x + x) 1                   is not root-reduced;
(3, 2 + 2)                       is root-reduced, not reduced;
(3, `x . 2 + 2)                  is reduced;
(3, `x . 4)                      is in normal form.
```

I.4.3.2 Semantics

The contraction rules of TALE are such that they form a regular combinatory reduction system in the sense of Klop [1980], for which he proved the Church-Rosser theorem. This implies that if an expression has a normal form, then it is unique. This gives a well defined operational semantics to TALE.

As for a denotational semantics, we believe that the ideas in Scott [1976] or Mac Queen et al. [1984] provide the necessary mathematical structure.

I.4.3.3 Some Implementation Issues

- Reduction strategies. There are expressions having a normal form but also with parts without a normal form. For example, even if E has no normal form the expression

```
if B then 2 else E fi
```

has 2 as normal form if B reduces to true. Therefore the order of reduction is important. This order will be given by a so called reduction strategy. If we are

interested in reducing until we find the normal form, then the strategy of taking always the leftmost outermost reducible expression (even if it is within some lambda expression) works. However we may be interested in other strategies. Let E be an expression of type (t | s) and F a function defined via a pattern match

F E = case E of F1 | F2 esac.

If the leftmost reducible expression is in E, then at the moment E has reached it root-reduced form, the whole case-expression becomes the leftmost reducible expression. Therefore, in a parallel implementation of TALE it may be convenient that the separate sequential reducers do not reduce to normal form but rather to (root-)reduced form. Such a reduction strategy is called 'lazy', since it does not reduce any term any further than to the point it knows to be absolutely necessary. In general, unless we have a way of terminating reduction processes that consume resorces without leading to the normal form, we can only start reducing a sub-term when we are sure it will be needed (or alternatively when we are sure that the reduction will terminate, but in general this is undecidable). If, as we require in TALE, the output of a complete program is of a data type built up from the primitive types without the function constructor (.->.), then the notions reduced form and normal form of the complete program coincide, and we will get the output also by a reduced form finding strategy.

Another strategy that is used, e.g. for LISP, is so-called 'eager' or 'applicative order' reduction. In this strategy, only innermost reducible expressions are contracted. In particular, the arguments Ai of a function application F A1 .. An are reduced to normal form before F is done. For TALE, eager evaluation is not appropriate, since it fails to find the normal form of e.g. hd(1:omega) if omega has no normal form. However we should emphasize that tabulation of arrays works in an easier way: if F 0 = 2 but F 1 has no reduced form, then

tab ((0,1)) : F bat) [0]

does not reduce to 2 by any strategy, hence has no normal form, since it is explicitly required that all components be reduced before an array can be formed at all. In fact it is preferable to deal with arrays in an eager way, since this allows a lot of paralellism.

- Graph reduction. One can implement a reduction like

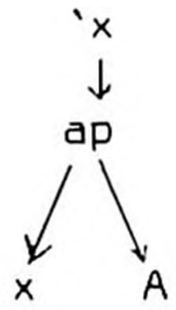
('x. ....x...x....) A -> ....A...A...

literally, i.e. by actually substituting A for x. This so called string reduction is rather expensive, since A may be large. It is better to use pointers:

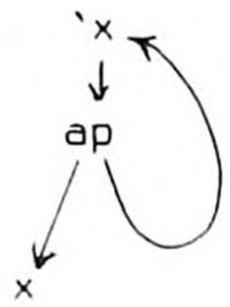
('x. ....x...x....) A -> ....○...○... A

After some reductions like this terms become rather involved graphs. Reduction implemented this way is called graph reduction.

A graph is also used to represent recursion. For example 'x.x A is represented as



But if A = rec a: 'x.x a, i.e. A -> 'x.x A, then this becomes



In this way cyclic graphs represent recursion.

-Nameless dummies. A reduction like

(`x.`y.E(x,y))(2*y) -> `y.E(2*y,y)

is not correct, since in the first expression the y in 2*y is a free (global) variable that becomes bound (local) in `y.E(2*y,y). This can be improved by renaming bound variables (alpha-reduction): (`x.`y.E(x,y))(2*y) is changed into (`x.`yy.E(x,yy))(2*y) which then reduces to `yy.E(2*x,yy). This renaming of bound variables is also rather expensive. Moreover, often some bugs appear in an implementation of this mechanism.

N.G. de Bruijn [1972] avoided this problem by representing a lambda term like

`x.`y. x (y x)

as

` ` 2 (1 2)

The first 2 refers to the fact that it is a variable that is bound '2 lambdas above'; etcetera. Another example

`x. x (`y. y x)

becomes

` 1 (` 1 2)

Now terms that are equal up to the names of bound variables are represented in the same way: `x. x and `y. y are both denoted as `1. A term with free variables like `x. x a b can be written as

.... c b a | ` 1 2 3

Since this notion uses variables as pointers, it is nicely compatible with graph reduction.

CHAPTER II

THE LANGUAGE TALE

In this chapter the language TALE, which stands for Typed Applicative Language Experiment, is described and defined. The main reason for defining a new language is that we find that no existing functional language, to our knowledge, sufficiently clearly displays the simple semantic elements from which functional algorithms may be built up. Also we have found no functional language that equally well supports arrays as it does linked data-structures (like lists, trees, etc.).

II.1 INTRODUCTION

TALE is an experimental functional language, that has not been implemented (nor do we have concrete plans for doing so). Apart from serving as a vehicle for illustrating functional programming in this paper, this language was intended to satisfy the following design goals:

1. High expressive power, enabling specification of (complex) computational algorithms in a forthright and mathematically transparent way.

2. Simple and unambiguous semantics.

3. Flexible, though not excessively abundant, syntactical forms, allowing a readable and concise format of expression.

4. Completeness of semantic primitives with respect to functional algorithms. This means that functional programs written in other languages should be translatable into TALE without the need to simulate conceptively simple operations by extensive combinations of operations. Clearly this criterion is not rigorously defined, but is interpreted in the sense that at least all commonly used data-constructions and e.g. choice-operations should be present.

For achieveing these external goals, as far as they are not a consequence of functional programming in general, we decided to pursue the following internal characteristics:

a. Simple and straightforward reduction semantics, based directly on the lambda calculus. All semantic additions (apart from explicit recursion) are directly related to either base values, or to some form of data-structuring, of which there are three kinds: cartesian product, disjoint union and array-formation.

b. A relatively simple, orthogonal polymorphic type system, using structural equivalence.

c. Possibility to state all operations and types explicitly. Omission, or more implicit forms are allowed only when the explicit forms can be unambiguously reconstructed.

d. Orthogonality. Language constructions are generally allowed in all situations where they would be meaningful, with no unnecessary exceptions. Also orthogonality is enhanced by using as much as possible the same concept for all related purposes. So for instance there is but one concept for passing information outward from an expression, namely yielding a (not necessarily reduced) value.

e. Definiteness. The language is defined accurately, if not formally. Most importantly this is the case for the semantics of language constructs.

It was recognised that there is a potential conflict between semantic simplicity and syntactic flexibility. Therefore we decided to separately define a 'core language', containing all semantics in their pure form. The rest of the language is described by 'extensions', providing more flexible and/or compact syntax for constructs of the core language, together with rules telling to which construct in the core language they are equivalent. In principle, it is intended that this replacement by constructs of the core language may be performed by the front-end of a compiler, the rest of the compiler dealing only with the core language. Moreover this approach should not give rise to an (unacceptable) efficiency penalty. Of course we cannot exclude the possibility of a back-end of a compiler supporting part of the full language directly, but that was not a point considered in designing the extensions. The translations from other languages mentioned in (4) can in principle be translations into the core language.

The language description is devided into the following steps:
- description of the core language: its syntax and semantics,
- description of the extensions, and their replacement rules,
- the initial environment (giving built-in functions, operator-, type- and priority-declarations etc.),
- the collected syntax of the language.

The last item is useful because every extension changes the syntax. The grammar given is however subject to the restriction, that only those constructs that are obtainable by repeatedly applying extensions to a valid construct in the core-language, are legal. This restriction is particularly relevant in case of pattern-matching declarations.

It may be noticed that TALE bears much superficial resemblance to Algol 68, though it is much less complicated. Indeed lots of concepts were borrowed from it, possibly in modified form. Also in case of design decisions with many acceptable choices, one resembling a choice made in Algol 68 was often made. Doing so hopefully leads to a certain coherence of style. We might suggest that in concept, though not in practice, Algol 68 is close to functional languages. Despite this relationship, TALE is of course semantically closer to other functional languages than to Algol 68.

Because TALE is designed to describe computational algorithms rather than to control computers, you will find no description of input/output operations. This is a deliberate choice, since we consider these operations to be imperative in nature, and not fit for expression by reduction semantics. The result of a program is simply the value it reduces to (if any). In an actual implementation, there should of course be some way to enter input data, form a function application of a precompiled function to it, and process in some way the result yielded. These actions could be performed by an interactive expression-evaluator, but might just as well be realised by making the compiled functions (together with the reducer) accessible as a function-library to other (possibly imperative) languages. We consider this interface to the real world as part of the programming environment, rather than as belonging to the programming language proper, as much as is the case with

text-editors, compilers, operating systems, and the like.

II.2 CORE LANGUAGE

The core of the language is a variant of the 2nd order typed lambda-calculus, introduced by Girard [1972] and Reynolds [1974]; see also Kahn et.al. [1984]. This calculus is extended to include the primitive data-constructions of cartesian product, disjoint union and array-formation, and including recursion expressions and recursive types. By the last two features, the language TALE is of the strength of the type-free lambda calculus, see Barendregt [1984], and therefore it can represent all computable functions.

II.2.1 Method of Description

In this section we collect general remarks on the language as a whole, and the way it is described. We note that, by convention, we use single-quotes ('') to group sequences of symbols that are subject of discussion, or for the introduction of technical terms.

II.2.1.1 Meta Syntax

The syntax is given in an extended BNF-format, with the following conventions:

- syntactic categories are enclosed in pointy brackets (<>)
- literally represented symbols are enclosed in double-quotes (""), the double-quote symbol itself being denoted by <quote>.
- an entity may be followed by a digit or a lowercase-letter (generally i,j..) without change of meaning, this is done to be able to talk about e.g. <expression>2 to identify a specific occurrence of <expression> in a rule. Lowercase letters are used when a entity is repeated to indicate the i-th repeated occurrence; they bear no relationship to "digited" (i.e. indexed by concrete numbers) entities.
- any other marks are part of the BNF meta-syntax, notably:
  - the symbol '::=' denotes a production rule of the context-free grammar
  - the symbol ':==' denotes a lexical production, giving only representation, while the left-hand-side is grammatically an indivisible symbol (or even just part of one). E.g. we might use this to indicate that a number is represented by a sequence of digits, without implying that these digits have an individual meaning
  - commas separate the syntactic entities, implying juxtaposition
  - vertical bars separate alternatives for a BNF-production rule (commas bind more strongly than vertical bars)
  - a full stop terminates a production rule
  - parentheses are used for grouping, forming an anonymous category
  - a question mark (?) following an entity means it is optional (so '"-"?' indicates an optional minus sign)
  - a star (*) directly following a entity means zero or more repetitions
  - a plus-sign (+) means one or more repetitions (so 'x+?' would be equivalent to 'x*' for any category x)
  - CHAIN is an infix operator, 'x CHAIN y' meaning 'x,(y,x)*'. Here y is usually a separator symbol; e.g. a sequence of <statement>s separated by semicolons would be noted '<statement>CHAIN";"'
  - +CHAIN is likewise, 'x +CHAIN y' meaning 'x,(y,x)+' (note that the seperator y occurs at least once, so that the category x is repeated at least twice)
  - LIST is a postfix operator, 'x LIST' meaning 'x CHAIN ","'
  - +LIST is likewise, 'x +LIST' meaning 'x +CHAIN ","'

II.2.1.2 Expressions and Types

As a starting point we give the productions for the main category <expression>. (Inside compound names of syntactic categories, the words 'expression' and 'variable' are contracted to 'expr' and 'var' respectively).

```
<expression>
    ::= <lambda expr> | <recursion expr> | <polymorphic expr> | <tertiary> .
<tertiary>
    ::= <function application> | <secondary> .
<secondary>
    ::= <formula> | <primary> .
<primary>
    ::= <specialisation> | <applied var> | <denotation>
      | <subscription> | <array update> | <descriptor transformation>
      | <error expr> | <enclosed expr> .
<enclosed expr>
    ::= "(",<expression>,")" | <tuple display>
      | <union display> | <case-of expr> | <case-in expr>
      | <array display> | <for expr> | <tabulation expr> .
```

This describes the main sub-categories of <expression>, and the general precedence structure. In the sequence <expression>, <tertiary>, <secondary>, <primary>, <enclosed expr>, we go from loosely-bound to tightly-bound expression forms. We will call 'expression' any production of any of these sub-categories of <expression>.

We now also give the production for <type>, which corresponds to the different possibilities for data-structuring.

```
<type>
    ::= <function type> | <recursion type> | <polymorphic type>
      | <primitive type> | <tuple type> | <union type> | <array type> .
<function type>
    ::= "(" , <type> , "->" , <type> , ")" .
<recursion type>
    ::= "rectype" , <type var> , ":" , <type> .
<polymorphic type>
    ::= "@" , <type var> , <type> .
<primitive type>
    ::= <base type> | <type var> .
<base type>
    ::= "char" | "int" | "real" .
<tuple type>
    ::= "*" | "(" , <type>+LIST , ")" .
<union type>
    ::= "(" , <type>+CHAIN"|" , ")" .
<array type>
    ::= "[" , <empty>LIST , "]" , <type> .

<type var>
    :== <bold identifier> .
<empty>
    :== .
<bold identifier>
    :== <bold letter>+ .
```

A type is a terminal production for <type>. There is an equivalence relation for types, involving recursion and polymorphic types, defined in the sequel. A type without free <type var>s corresponds to a domain of values. The base types char, int, real correspond respectively to the domains of ASCII-characters, integers and real numbers (in some floating point representation).

The other kinds of types correspond to domain-theoretic constructions:
- A <function type> '(t1->t2)' corresponds to the domain of computable functions from t1 to t2,
- A <tuple type> '(t1,..,tn)' corresponds to the cartesian product t1*..*tn (and "*" corresponds to the cartesian product of 0 factors, having 1 element),
- A <union type> '(t1|..|tn)' corresponds to the disjoint union t1+..+tn, and
- An <array type> '[,..,]t' with (n-1) commas, corresponds to the domain of maps from some finite block in int^n to t (i.e. of linear arrays, matrices etc. as n=1,2,...).

Recursion types are used to describe types which contain themselves as subtypes, so for instance 'rectype t:(*|(int,t))' corresponds to a solution of the domain equation D={ () }+int*D. In fact recursive types should be viewed as directed graphs rather than as tree-structured objects, the indroduced type variable serving as a label for making cycles in the graph. Type equivalence for these cyclic graphs is structural, that is types are equivalent if they look the same descending to any depth from the root; this makes 'rectype t:(int,(int,t))' equivalent to 'rectype t:(int,(int,(int,t)))'. The type rectype t: t is illegal.

Polymorphic types are "general" types that may be specialised to other ones. Specialisation involves substituting a <type> for the <type var> of a <polymorphic type>, so e.g. the type @x(x->x) may be specialised by int to (int->int). The type @x(x->x) is the type of the (polymorphic) identity function, and above specialisation shows that that function may be applied to an integer, yielding an integer. So the symbol '@' may be read as universal quantifier for types ("for all .."), binding any free occurrences of the <type var> in the <type> following it; and systematic renaming of such type variables gives equivalent types.

II.2.1.3 Grammatical Attributes

Any expression has three attributes:
- a static environment, which describes identifiers, operators, type variables etc. that are in scope at that point, and their static properties (e.g. type, priority);
- a type, which describes the domain in which the value, if any, represented by this expression will lie;
- a strength which is either weak or strong, which tells whether (strong) or not (weak) a specific type is required by the context. It can be checked that the rules for types indeed admit only a single type in any strong context.

Static environment and strength are inherited attributes, determined by the context. On the other hand type is a synthesised attribute, determined by the expression itself, though some information from the environment may be needed to determine it. With each production rule there is a corresponding propagation of attributes, from left-hand-side to right-hand-side for derived attributes, the other way round for synthesised attributes. This propagation is specified by relations that are required to hold between attributes of the entities appearing in that rule. In relations for types, the '=' sign indicates equivalence, in relations for environments (env), only the set of entries is specified, their properties being assumed to follow. Also environments have the usual layered structure, allowing a search for the most recently added entry of some kind, so a relation 'new-env = old-env + {new entries}' must be interpreted as adding a new layer to old-env forming new-env. In the core language, environment changes only occur in <lambda expr>s, <recursion expr>s and <polymorphic expr>s, so we can save ourselves some text by omitting the rather trivial relations for the environments in all other production rules: the environments of all occurring expressions must be the same. As an example of these relations, any production for which the RHS is a single syntactic category, as well as the production '<enclosed expr> ::= "(",<expression>,")"' is semantically neutral, and attributes are propagated unchanged, which may be expressed as 'type(<enclosed expr>)=type(<expression>)',

'strength(<expression>)=strength(<enclosed expr>)' etc.
Furthermore it may be noted that the relations for type can, in case of strong contexts, also be used to determine the type that expression is required to have.

Apart from expressions, other syntactic entities may have attributes, so e.g. a <var plan> also has a type, (which happens to be an inherited attribute in this case) thus giving all variables a type at their defining occurrence.

II.2.1.4 Lexical Conventions

We note the following lexical conventions in the language. There are at least 2 alphabets, that of <letter>s and that of <bold letter>s. In this paper we write <bold letter>s as underscored small letters for aesthetical reasons, but in practice they will probably be capital letters. Sequences of <bold letter>s appearing litterally in the syntax, with the exception of the <operator>s "descr" and "within", are keywords of the language. Keywords are reserved symbols, and may not be used as <bold identifier>s.

<letter> :== "a" | "b" | .. | "z" | "_" .
<bold letter> :== "a" | "b" | .. | "z" .
<digit> :== "0" | "1" | .. | "9" .

A 'separation' consists of a sequence of one or more blank speces, changes to a new line or page, or <comment>s. A separation may not occur within a single grammatical symbol, but is always allowed between two symbols. Including a separation between two symbols is even mandatory, whenever juxtaposing the symbols would make two <letter>s, two <bold letter>s, two <digit>s, two <quote>s, or a <digit> and a <letter> adjacent. Separations appear only for reasons of disambiguation and readability, and they bear no meaning. Comments are enclosed in curly brackets ({}) and may contain any characters, but if curly brackets appear they should be properly matched.

<comment>
    :== "{" , (<any ascii character but curly brackets>|<comment>)* , "}" .

II.2.1.5 Reduction Semantics

We now briefly describe the process of reduction, which will be used to give the semantics of the language. To do this, we use the concept of 'terms', which are the objects being manipulated in our semantic model (it is not required that an implementation uses entirely similar objects (e.g. it might use graphlike and cyclic structures), as long as it produces the same data-values as results). Terms take an intermediate position between expressions and data-values (the latter of which are assumed to have meaning independent of our semantics, like for instance numbers or characters). Actually, terms include data-values as a special case, but there are also not (completely) evaluated terms, corresponding to more general kinds of expressions. Like expressions, terms generally have a nested, tree-like composition. Any expression 'denotes' some term, though not all terms are denotable (e.g. negative numbers and arrays are not). Terms don't display certain syntactic details present in expressions (like parentheses), while on the other hand they do exhibit some internal details that expressions do not (like the representational precision of real numbers). Also terms do not carry any type information. In general, the term denoted by an expression is obtained as follows. Starting with the syntax tree of the expression, we first obtain the abstract syntax tree by short-circuiting any nodes with a single descendant (like resulting from '<expression>::=<tertiary>'). Then we prune all parts pertaining only to the type system (i.e. removing all <type>s and replacing <polymorphic expr>s and <specialisation>s by their constituent expressions). Also, primitive expression forms, like <denotation>s, are replaced by the terms they directly denote. For the sake of brevity, we shall usually omit the phrase 'the term denoted by' (some expression).

Reduction now is the process of repeatedly replacing (sub)-terms by other ones, according to rules given, until no further replacement is possible. This final form is called a 'normal form', and is a data value representing the result of the program. Since our reduction system has the Church-Rosser property, this normal form, if it exists, is unique. Hence the order of reduction, insofar as it is not fixed by the reduction rules themselves, is rather arbitrary. We do require however that a strategy for selecting reductions is adopted, that guarantees that whenever a normal form exists, it will eventually be reached.

A term is 'reduced' if it contains no reducible subterms, except possibly within contained <lambda expr>s, so a term is reduced when it is either

- "error",
- a base value,
- a tuple, all of whose components are reduced,
- a union, whose component is reduced
- an array (whose components are always reduced),
- a function (i.e. either a <lambda expr> or a built-in function).

Clearly a reduced term that contains no functions is a normal form. As we require that the type of the whole program (i.e. the expression denoting the term that is to be reduced) contains no <function type>, it suffices to reduce the term it denotes to reduced form. Actually the reduced form is the furthest form of reduction we will require for any term (reduction inside <lambda expr>s is legal but never necessary). However, there is a less strict concept we will use in certain places: a term is 'root-reduced' when it is "error" or a base value, tuple, union, array, or function.

We now proceed to the semantically more interesting production rules, grouping them by the data-contructions involved, in the order functions, recursion, polymorphism (no data-constructions), base values, tuples, unions, arrays.

II.2.2 Functions, Recursion and Polymorphism

Functions are the most powerful elements in the language. There strength is enlarged by the possibilities of recursion and polymorphism. Recursion and polymorphism are in fact not restricted to use with functions only, though it does form the most important use of them.

II.2.2.1 Functions

Functions are the basic algorithmic units in the language, and a function of type (d->c) is an algorithm for computing a value in the (co)domain c from an argument in domain d. Applying a function to an argument is the only way to access the algorithm. Functions themselves are perfectly legal values and may be handled like any other value. The only restriction, mentioned above, is due to the fact that an algorithm only has meaning within the framework of the language semantics: a function can never be (part of) the ultimate result of a functional program. We use the word data to indicate any values that are not, nor contain any, functions; these are the ones that may be the ultimate result of a program. So you can't expect that a function you created will be printed as output. This saves implementers from the embarresment of having to produce a textual representation of e.g. their floating-point multiplication function. More importantly it eliminates the need to reduce inside <lambda expr>s. This allows functions to be represented internally in a form that bears no direct resemblance to the piece of program-text defining them, as long as the implementation is able to perform the specified algorithm. In particular, an implementation may use combinators. Therefore the substitution-semantics given here for function-application should only be considered as a model defining the desired effects, rather than imply that substitution should be the basic operation of the reducer.

Functions are either built-in or they are formed by <lambda expr>s. Built-in functions (like integer addition etc.) are maps from closed terms to closed terms. Built-in functions themselves are also terms, though they cannot be denoted. However, they are bound to certain identifiers in the initial environment, so that they may be accessed by <applied var>s for which the corresponding term will be substituted prior to reduction.

Lambda Expressions - A <lambda expr> specifies an algorithm by means of substitution, like in the lambda-calculus: a formal parameter is introduced, and an expression for the result is given, into which actual arguments are to be substituted. A difference with the lambda-calculus is that functions may be polyadic (i.e. have compound parameters), but the related issues are discussed in the section on tuples.

```
<lambda expr>
    ::= "`" , <formal> , ("."|"->") , <expression> .

type(<lambda expr>) = ( type(<formal>) -> type(<expression>) ),
strength(<expression>) = strength(<lambda expr>),
env(<expression>) = env(<lambda expr>)+{<variable>|<variable> in <formal>}.

<formal>
    ::= <typed formal> | <var plan> .
<typed formal>
    ::= <type> , <var plan> .

type(<var plan>) = type(<typed formal>) = <type>.
```

So lambda expressions resemble those of the lambda-calculus, the greek letter lambda being replaced by "`", the "." optionally by "->", and with opportunity to specify the type of the parameter. In fact the <formal> must be a <typed formal> if the <lamba expr> is weak, so that the type of the <var plan> can be deduced in any case. We defer a further description of <var plan>, but note that there is indeed an alternative '<var plan> ::= <variable>|...' (so the ordinary lambda-expression is contained as a special case). The <var plan> of a <formal> contains a set of <variable>s; containment is denoted by '<variable> in <formal>'. There are no semantics for <lambda expr>s, since they are root-reduced.

Variables - At this point we insert the description of variables, which are used inside <lambda expr>s to indicate (a part of) the parameter (they are used in <recursion expr>s as well). They have no semantics of their own and serve merely as a target for substitution. Applied occurrences of variables will always be denoted <applied var>, <variable> being used for binding occurrences.

Of course TALE has static binding. Therefore, in describing reduction by substitution we will have to avoid name clashes. This is a subtle point in the description of any reduction system using substitution; usually something is said about renaming variables whenever there is a danger of name clash. We take a slightly different viewpoint that seems a bit more natural and definite than the usual one, and effectively comes down to renaming whenever a copy of a term is made. To avoid any impression that the reduction behavior is subtly affected by the choice of <variable> names (i.e. other than via static identification) we detatch <variable>s and <applied var>s from their textual representation after having established their initial identification. Without names no name clashes, but of course we now have to carefully prescribe how identification will evolve during reduction, most importantly during substitution. So we proclaim that each occurrence of a <variable> denotes a distinguished term, which is an atomic object called a 'binder', and we will take care that no binder will ever occur more than once in a term. Likewise, each occurrence of an <applied var> denotes a distinguished term, which is also an atomic object called a 'tag'. Each tag will 'identify' a unique

binder, and there is an infinite supply of distinguished binders, and of tags identifying any binder. During "ordinary" reduction, tags and binders behave like any other term, but we can make a 'fresh copy' of a term as described below; this happens only in the process of substitution. A 'fresh copy' of a term T is created as follows. Any binder B occurring in T is replaced by another binder f(B) that is not used anywhere else, and any tag I in T that identifies a binder B in T is replaced by a tag f(I), so that f(I) identifies f(B). All other tags (i.e. identifying a binder outside of T) remain unchanged.

We establish the initial identification of tags and binders as follows. The tag denoted by an <applied var> identifies the binder denoted by the <variable>, that is the newest <variable> in the environment of that <applied var> that has a textual representation identical to it. We also say that the <applied var> identifies the corresponding <variable>. To make the rule above unambiguous, we require that all <variable>s in one layer, (i.e. occurring in one same <formal>), must differ textually. The <applied var> has the type of the <variable> it identifies.

<variable> :== <identifier> .
<applied var> :== <identifier> .
<identifier> :== <digit>* , <letter> , (<letter>|<digit>)* .

So <identifier>s must contain at least one <letter>, to distinguish them from <integer denotation>s.

Function Applications - Functions are used in <function application>s. The syntax is

<function application>
    ::= <tertiary> , <primary> .

type(<tertiary>) = (type(<primary>) -> type(<function application>)),
strength(<tertiary>) = weak,
strength(<primary>) = strong.

So function application is denoted by juxtaposition, the <tertiary> being the function (call it F), the <primary> the argument (call it A). The semantics for it depends on whether F yields a <lambda expr> or a built-in function, so it is required that F be reduced to one of these forms first.

If F is a built-in function, reduction of the <function application> may require reduction to be performed upon A first, depending on that built-in function, and then the <function application> reduces to (the map) F applied to A (so this is real function application, in the mathematical sense).

If F is a <lambda expr>, containing a <var plan> P and <expression> E, a set S=Bind(P,A) of 'bindings' is formed according to P and A; this is described in the section on tuples and may require some reduction of A (in case P is just a binder B (denoted by a plain <variable>), S will be a singleton set { (B,A) }). In any case S will consist of a set of pairs (Bi,Ei), where the Bi are different binders, and the Ei are terms. The <function application> is now reduced to the result (written 'E [P:=A]') of the following substitution performed upon E: for each occurrence of a tag (denoted by some <applied var>) in E, that identifies some binder Bi in S a fresh copy of the corresponding term Ei is substituted. We may indicate this rule schematically (omitting types) by

(`P->E) A => E[P:=A]

In case P is (denoted by) a single <variable>, this is just the Beta-rule of the lambda-calculus.

II.2.2.2 Recursion Expressions

Recursion expressions provide an explicit way to specify terms that contain (terms equivalent to) themselves. This is achieved by substituting the whole term for occurrences of a certain <applied var> within itself. The syntax is:

```
<recursion expr>
    ::= "rec" , <type>?  , <variable> , ":" , <expression> .

type(<recursion expr>) = type(<variable>> = type(<expression>) = <type>,
strength(<expression>) = strong,
env(<expression>> = env(<recursion expr>) + {<variable>}.
```

As with the <lambda expr>, <type> must be present if the context is weak. The semantics of a <recursion expr> R with binder B (denoted by the <variable>) and <expression> E, is that R reduces to 'E [B:=R]'. Schematically (omitting types):

rec V: E => E [V := rec V: E]

Substituting the whole term is the best we can do, given that terms are tree-structured, but it may necessitate repetition of this process when the substituted term is needed. If an implementation uses graph-structures, cyclic graphs may be used with gain of efficiency, especially in case of recursively defined data objects.

II.2.2.3 Polymorphism

Polymorphic Expressions - Polymorphic expressions allow for the introduction of new <type var>s, indicating that a certain expression would be properly typed regardless of which type is substituted for that <type var>; this is most useful in the case of <lambda expr>s where the <type var> is used in the <type> of its <formal> and possibly in its <expression>. The polymorphic expression itself gets a polymorphic type (with '@'). The syntax is:

```
<polymorphic expr>
    ::= "%" , <type var> , <expression> .

type(<polymorphic expr>) = @ <type var> type(<expression>),
strength(<expression>) = strength(<polymorphic expr>),
env(<expression>) = env(<polymorphic expr>) + {<type var>}.
```

We see that environments may contain entries for <type var>s; there are however no properties associated to such a <type var> (though it may be used in types associated to newer entries), which guarantees that type-correctness in no way relies on properties of a type that could be substituted for the <type var>. We have intendedly used a symbol "%" different from "@" used inside types to mark the different uses; the "%" symbol is related to the "`" symbol in <lambda expr>s (binding a <type var> rather than ordinary <variable>s), whereas "@" in types may be interpreted as universal quantification. There is also a difference of using <type var>s bound by % or @: @-bound <type var>s are local to the type they occur in, while %-bound <type var>s may be free in the type of an expression, provided that <type var> occurs in the environment of that expression.

We impose 2 restrictions on the use of polymorphic expressions:

- The <type var> used may not already occur in the environment of the polymorphic expression; this prevents the occurrence of any <applied var>, whose type contains a <type var> referring to a (%-)binding which has been overruled at that point, effectively prohibiting that type to be written down. Of course this restriction is easily met. Actually this restriction holds for any newly introduced <bold identifier>, so we may also not re-introduce an existing <type var> by e.g. rectype.

- We require that the <type var> actually occurs freely in the type of the <expression> following it. As a consequence any polymorphic type must contain at least one use of the @-bound <type var>, and so all different substituions to a polymorphic type yield different types. This will be a pleasant property in the sequel, and it is hard to imagine a useful program being ruled-out by this restriction.

Specialisations - Expressions with polymorphic type have to be explicitly instantiated by a type. This happens in <specialisation>s:

```
<specialisation>
    ::= <primary> , "$" , <type> .
type(<specialisation>) = specialise(type(<primary>),<type>),
strength(<primary>) = weak.
```

Here 'specialise' is a meta-function on types, defined only if the first parameter type is a polymorphic type, with 'specialise(@x T,t)' meaning the result of substituting t for x in T. Note that this substitution only affects the type of the <specialisation>, and is of no influence upon the typing of the constituent expression.

Since there are no terms corresponding to either <polymorphic expr>s or <specialisation>s, we need give no semantics. However we might have safely assumed, like in the 2nd order lambda-calculus, that the types are carried around, and that reducing a specialisation actually involves substituting types; this would not infringe type-correctness (though it might produce well-typed programs from ill-typed ones). Doing so may be useful in proving that our type-system is correct, i.e. no ill-typed terms can be formed during reduction.

II.2.3 Base Values

The base values form the primitive data items to be manipulated. There are 3 sets of base values: characters, integers and reals. Base values may be specified by <denotation>s. Furthermore there is a number of built-in functions operating upon them, that are given in the initial environment.

II.2.3.1 Denotations

```
<denotation>
    ::= <character denotation> | <integer denotation> | <real denotation> .
<character denotation>
    :== "'" , <ascii character> .
<integer denotation>
    :== <digit>+ .
<real denotation>
    :== <digit>+ , "." , <digit>* , ( "e" , "-"? , <digit>+ )? .
type(<character denotation>) = char,
type(<integer denotation>) = int,
type(<real denotation>) = real.
```

Denotations denote the values they suggest: a <character denotation> denotes the <ascii character> it contains, an <integer denotation> denotes the integral value it decimally represents, and likewise a <real denotation> denotes a real value, the number following the "e" indicating a power of 10.

II.2.3.2 Case-In Expressions

There is only one other syntactic form involving base values:

```
<case-in expr>
    ::= "case" , <expression>0
      , "in" , <expression>iLIST
      , "out" , <limb>
      , "esac" .
<limb>
    ::= <expression> .

type(<expression>0) = int,
type(<case-in expr>) = type(<expression>i),
type(<limb>) = ( int -> type(<case-in expr>),
strength(<expression>0) = strong,
strength(<expression>i) = strength(<limb>) = strength(<case-in expr>).
```

The category <limb> is added only for defining certain extensions. The <case-in expr> specifies a multi-way choice using an integral value. A <case-in expr> 'case I in E0 , .. , En out F esac' reduces as follows: first I must reduce to an integral value, say i, then if 0<=i<=n the <case-in expr> reduces to Ei, otherwise it reduces to the <function application> 'F i'. We have schematically:

```
case i in E0,..,En out F esac    =>    Ei     (if 0<=i<=n)
                                       F i    (otherwise)
```

Note that the first expression following 'in' corresponds to a value of 0 . Furthermore, that first expression is the only one than must always be present, so that if you just want to test if i equals zero, you may write 'case i in E out F esac'.

II.2.3.3 Error Expressions

Not strictly a base-value, but important anyway, is "error", provided to explicitly indicate that something has gone wrong, and the program is not able to deal with it. It is therefore not possible to do "error-handling" using "error"; to have a controlled continuation in an exceptional condition, a union-value should be used whose variant indicates the exception. Nevertheless, it might be be desirable to export some information when "error" is yielded by a program, enabling a quicker localisation of the offending program-part; therefore we provide a locus for this information, though it has no official status in the semantics, and it is up to the implementation to decide what to do with it.

```
<error expr>
    ::= "error" , <enclosed expr> .

type(<error expr>) = @t t,
strength(<enclosed expr>) = weak.
```

The <enclosed expr> is the aforementioned information to be exported; an implementation might print the result of reducing it on an error output channel after having notified the error. There is a unique term "error", that any <error expr> represents, and that is root-reduced. The semantics of "error" actually involves that of a lot of constructs, but it is left out of their description for clarity. Whenever the reduction of any term T is said to require the reduction of a term S (which may or may not be a sub-term of T) to any kind of root-reduced form, but this reduction of S leads to "error" instead, then T also reduces to "error". Note that there may be more than one term S required by T, and in this case any of them may cause T to reduce to "error"; consequently an indication by an implementation of the offending expression might be non-deterministic. The

propagation of "error" thus defined behaves exactly like the propagation of non-termination (i.e. failure of an expression to come to root-reduced form), which of course needn't be explicitly stated. The difference between both ways of a program to fail to deliver a result, is that an implementation should at least notify the yielding of "error", while non-termination can only be detected by a time-out of your patience, as it happens to be an undecidable predicate.

II.2.4 Tuples

The simplest way to combine values into a composite value is forming tuples. Given domains D1 , .. , Dn there is a domain isomorphic to the cartesian product of the domains Di, its elements correspond to sequences v1,..,vn, with vi in the domain Di; we call these elements n-tuples and write (v1,..,vn). The 'components' vi of a tuple can be unreduced terms. We exclude 1-tuples since they are of no use, and would give rise to syntactic ambiguity; there is however a 0-tuple type '*'.

II.2.4.1 Tuple Displays

Tuples are formed by simply writing down the component expressions:

```
<tuple display>
    ::= "()" | "(" , <expression>i+LIST , ")"

type( "()" ) = *,
type(<tuple display>) = ( type(<expression>i)+LIST ),
strength(<expression>i) = strength(<tuple display>).
```

A <tuple display> denotes the tuple of its constituent <expression>s ("()" denotes the 0-tuple), which is a root-reduced term.

II.2.4.2 Multiple Binding

There is no form of expression that directly selects a component from a tuple; rather, this is achieved by polyadic functions, i.e. <lambda expr>s whose <var plan> contains more than one <variable>. Therefore we give first the production rules for <var plan>:

```
<var plan>
    ::= <variable> | <compound plan>
      | <variable> , "==" , <compound plan> | "-" .

type(<variable>) = type(<compound plan>) = type(<var plan>).

<compound plan>
    ::= "()" | "(" <var plan>i+LIST ")" .

type("()") = *,
type(<compound plan>) = ( type(<var plan>i)+LIST ).
```

The semantics of polyadic functions was given largely in the section on functions, it suffices to specify the set Bind(P,A), of bindings formed according to a <var plan> P and a term A. We give an inductive definition, following the structure of P.

- If P="-", then Bind(P,A) = {} (the empty set);
- if P is a binder (denoted by some <variable>), then Bind(P,A) = { (P,A) };

  if P is a <compound plan> or '<variable>==<compound plan>', then it is required that A be reduced to a tuple, say '(A1,..,An)', where n will equal the number of constituent <var plan>s of the <compound plan>.

- if P is '(P1,..,Pn)', then Bind(P,A) = Bind(P1,A1)+..+Bind(Pn,An)
  (here '+' denotes union of sets);
- if P is 'B==(P1,..,Pn)', then
  Bind(P,A) = {(B,A)}+Bind(P1,A1)+..+Bind(Pn,An)
  (here B is a binder (denoted by some <variable>)).

We give two examples to illustrate this definition:

```
Bind( (x,y,z) , (A,B,C) ) = { (x,A) , (y,B) , (z,C) }
Bind( (x,y==(z,-)) , (1,((2,3),4)) ) = { (x,1) , (y,((2,3),4)) , (z,(2,3)) }
```

II.2.5 Unions

Union values represent a choice between a finite number of alternatives. Given domains D1 , .. , Dn there is a domain isomorphic to the disjoint union of the domains Di, its elements correspond to pairs <i,v>, where 1<=i<=n, and v is in the domain Di; we call these elements n-unions. We write pointy brackets to distinguish the n-union <i,v> from the 2-tuple (i,v), which is quite a different value. In the union <i,v>, i is called the 'discriminator', v the 'component'; the discriminator will just be a (small) intergral number, but the component can be an unreduced expression (like is the case with tuples). 1-unions are excluded for the same reason 1-tuples are; 0-unions don't exist.

II.2.5.1 Union Displays

To specify a union-value of type (t1|..|tn), one must specify a discriminator i (1<=i<=n), an expression of type ti, and furthermore all other types tj (j≠i). This is accomplished by a <union display>:

```
<union display>
    ::= "(" , <expression> , ( "|" , <type>? )+ , ")"
      | "(" , ( <type>? , "|" )+ , <expression> , ( "|" , <type>? )* , ")" .
```

So a <union display> consists of a parenthesised list of at least 2 items separated by vertical bars, one item being an <expression>, all others being a <type> or empty. Calling the i-th item (or its type, if it's the <expression>) ti, we have:

```
type(<union display>) = ( ti+CHAIN"|" ),
strength(<expression>) = strength(<union display>).
```

The <type>s may be omitted only if the <union display> is strong. If the i-th element of a <union display> U is an <expression> E, then U denotes the n-union <i,E>, which is a root-reduced term.

II.2.5.2 Case-Of Expresssions

To use a union-value, one must first inspect the descriminator. This happens in the <case-of expr>:

```
<case-of expr>:

<case-of expr>:
    ::= "case" , <expression> , "of" , <caselimbs> , "esac" .
<caselimbs>
    ::= <limb>i+CHAIN"|" .

type(<expression>) = (t1|..|tn),
type(<limb>i) = ( ti -> type(<case-of expr>),
(for some sequence of types t1,..,tn)
strength(<expression>) = weak,
```

strength(<limb>i) = strength(<case-of expr>).

Like the <case-in expr>, the <case-of expr> specifies a multi-way choice, using a union instead of an integral value. A <case-of expr> 'case U of F1 | .. | Fn esac' reduces as follows: first U must reduce to an n-union, say <i,E>, then the <case-of expr> reduces to the <function application> 'Fi E'. This can be written schematically

case (|..| E |..|) of F1 | F2 | .. | Fn esac => Fi E
(where i is the position of E in the <union display>)

II.2.6 Arrays

Arrays are aggregates of many components of the same type, which are selected by integers. Arrays allow for operations that uniformly apply to all components to be efficiently realised. In order to enable efficient handling, we require that components of an array are always reduced. This limits the orthogonality of the language: data structures that have no reduced form, e.g. infinite ones, cannot be collected in arrays; however one might encapsulate such structures in <lambda expr>s, preventing an attempt to reduce them. Arrays may be useful in the fields of numerical computation, statistics and data-base handling. Their use is somewhat unrelated to the particular features of functional programming, which explains why they are often left out of functional languages. Indeed it is more complicated to include arrays than tuples or unions. Alternatives have to be given for the classical imperative approach, updating single elements at a time, forcing sequentiality. Emphasis lies on powerful operations, rather than simple flexible ones, thus allowing the capabilities of the underlying machine to be fully used.

II.2.6.1 Basic Operations

Components of an array are selected by a number of integral indices, this number is called the 'dimension' of the array. An array type is characterised by a dimension n and a component type t, and is written [,..,]t with n <empty>s separated by (n-1) commas between the square brackets; for convenience we temporarily make the convention of writing this as [n]t. An array of this type is specified by the following data:

- an n-tuple ((l1,u1),..,(ln,un)) called the 'descriptor' of the array, of pairs of integers, called 'bound-pairs'. The descriptor specifies a block of values in int^n, namely the cartesian product of the intervals Ij = { m | lj<=m<=uj }.
- for each sequence 'i1,..,in' such that each ij is in Ij (we say that the 'index' 'i1,..,in' is 'within' the descriptor), a 'component' (called 'the component at' 'i1,..,in' is given, which is a reduced term in the domain corresponding to t.

Tabulation Expressions - The most general array-forming expression is the <tabulation expr>, that specifies the descriptor and a function computing the components:

<tabulation expr>
    ::= "tab" , <expression> , ":" , <limb> , "bat" .

type(<tabulation expr>) = [n]t,
type(<expression>) = (int,int)^n,
type(<limb>) = (int^n -> t)
(for some number n>=1 and type t),
strength(<expression>) = strength(<limb>) = strength(<tabulation expr>).

Here the notation x^n is used to indicate the n-tuple type, all of whose component types are x (if n=1, then it indicates x). The <expression> gives the descriptor, the <limb> computes the components.

A <tabulation expr> 'tab D : F bat' reduces to an array R defined as follows: first D must be reduced, yielding a descriptor; R has descriptor D, and for each index 'i1,..in' within D, the corresponding component of R is the term obtained by reducing the <function application> 'F (i1,..,1n)' to reduced form. So we have e.g.:

tab (1,n) : F bat = [[F 1 , F 2 , .. , F n]]

Note that reduction of a <tabulation expr> may require the independent reduction of a lot of other terms; this clearly gives opportunity for a parallel implementation to create parallel tasks, as will also be the case with <for expr>s.

Subscriptions - For retrieving a component from an array, we may use a <subscription>.

```
<subscription>
    ::= <primary> , "[" , <expression>iLIST , ext , <limb> , "]" .
```

Let n be the number of <expression>s in <expression>iLIST:

```
[n]type(<subscription>) = type(<primary>),
type(<expression>i) = int,
type(<limb>) = ( int^n -> type(<subscription>) ),
strength(<primary>) = strength(<limb>) = strength(<subscription>),
strength(<expression>i) = strong.
```

The <limb> defines the value if the index is out of bounds, it will often be a function always yielding "error" . A <subscription> 'A[i1,..,in ext F]' reduces as follows: first it is required that A is reduced (to an array) and also all the ij (to integers) (1<=j<=n). If the index 'i1,..,in' is within the descriptor of A, then the <subscription> reduces to the component of A at that index, otherwise it reduces to the <function application> 'F (i1,..,in)'. So we have e.g.

```
[[ A1 , .. , An ]] [i ext F]    =>    Ai      (if 0<=i<=n)
                                      F i     (otherwise)
```

Formulae: descr and within - To retrieve the descriptor from an array the <operator> "descr" is used. We therefore insert here a brief description of <formula>e, which are used in a limited way in the core language, but will be subject to extensions. The following cumbrous production rules are introduced to provide hooks for extensions, that give a more comprehensive treatment of <formula>e.

```
<formula>
    ::= <monadic formula> .
<monadic formula>
    ::= <monadic operator> , <monadic operand> .
<monadic operand>
    ::= <monadic formula> | <primary> .

strength(<monadic operand>) = weak.
```

The type rules for <formula>e are somewhat different than for other constructs. The following rule describes the general situation in the full language, but it applies nicely to the operators "descr" and "within" in the core language. The fact that these two operators need a type rule that not expressible by a single (polymorphic) type, is the reason they are not built-in functions.

In an environment, for each <operator> a set of legal 'operand types' is given, and a mapping associating them to 'result types'. Then in a <monadic formula>, it is required that type(<monadic operand>) is a legal operand type for the <monadic operator>, and type(<monadic formula>) is the associated result type. (A similar rule applies to <dyadic formula>e, but is not needed in the core language. Note that if the operand of a monadic formula is a 2-tuple like in '+(2,3)' it may also be written as a dyadic formula viz. '2+3'. Therefore, in the full language, "within" may be used as dyadic operator.)

We now proceed with arrays:

<monadic operator>
    ::= "descr" | "within" .

Legal operand types for "descr" are all array types '[n]t'; the associated result type is '(int,int)^n'. Legal operand types for "within" are '(int^n,(int,int)^n)' for any n>=1, the associated result type is '(*|*)' (i.e. Boolean).

Reduction of the <formula> 'descr A' requires that A be reduced to array first, the <formula> then reduces to the descriptor of A. So:

descr [[3 , 7 , 2 , 0]] = (1,4)

Reduction of the <formula> 'within A' requires that A be reduced to a tuple ((i1,..,in),D) first, the <formula> then reduces to either <1,()> (i.e. to true) when the index 'i1,..,in' is within the descriptor D, or else to <2,()> (i.e. to false) (we wrote '()' for the sole element of the cartesian product of 0 factors). So (using the full language):

(2,3) within ((1,2) , (2,20)) = true ; 0 within (1,0) = false

II.2.6.2 For Expressions

The expressions presented above suffice in principle for all manipulations with arrays; however powerful operations for building new arrays from old ones are useful. For this the most important one is the <for expr>, providing a substitute for the classical 'do-loop', but in a parallel way.

<for expr>
    ::= "for" <generator>iLIST , ":" , <limb> , "rof" .

Let m be the number of <generator>s in the LIST,
let type(<generator>i) = [li]ti
(for sequences li of integers and ti of types),
and let k be the sum of the li (1<=i<=m);
type(<limb>) = ((int^k,(t1,..,tm)) -> t),
type(<for expr>) = [k]t
(for some type t),
strength(<limb>) = strength(<for expr>).

Now, focussing on a single <generator> (in the following, n and l may vary between <generator>s):

<generator>
    ::= <expression>jCHAIN"||" .

Let n be the number of <expression>s in the CHAIN,
and let type(<expression>j) = [l]tj
(for one same integer l, and sequence of types tj (1<=j<=n));
type(<generator>) = [l](t1,..,tn),

strength(<expression>) = weak.

(Note that types like '(t1,..,tn)' may not be tuple-types if n=1; correspondingly some "tuples" mentioned in the sequel may actually not be tuples, when the number of components equals 1).

As a first explanation: the <expression>s in <generator>s must all yield arrays, which will be traversed over all their elements; those within one <generator> are traversed simultaneously (i.e. at equal indices), those in different <generator>s are traversed independently. During this traversal (which may by the way be done in parallel), all indices used are collected into a tuple, corresponding to an index for the array that will be the result of the reduction; all components found are collected into another tuple, and the <limb> is used to compute a component for the result array from the pair of these tuples.

More formally, a <for expr> '<u>for</u> g1,..,gm : F <u>rof</u>' reduces to an array A defined as follows: each of the gi is 'traversed' giving (intermediate) arrays Gi (see below); a descriptor D is formed by 'catenating' the descriptors Di of the Gi (i.e. their bound-pairs are taken, in order, and rearranged to a single tuple of bound-pairs). The descriptor of A is D, and we proceed to define the component of A at an arbitrary index 'h1,..,hk' within D. According to the formation of D, this index can be split up into indices within the Di, call these h~i (for 1<=i<=m), and let ai be the component of Gi at the index h~i; then the component of A at 'h1,..,hk' is the result of reducing the <function application> 'F ((h1,..,hk),(a1,..,am))' to reduced form.

Traversing a <generator> 'E1||..||En' gives an array A as follows: first it is required that the Ej (1<=j<=n) all be reduced to arrays, and all descriptors of the Ej must be equal, say they are all D, (otherwise the reduction requiring the traversal yields "<u>error</u>"); then A has descriptor D and and its component at an index h~ within D is (c1,..,cn) where cj is the component of Ej at h~.

It is suggested that an implementation not follow this description in an interpretative fashion, but rather compile specific code for each <for expr>; when used in combination with other array-manipulating expressions this approach can be taken even a step further.

We illustrate the <for expr> with a few typical examples. We use the full language for readability; amongst others this allows us to omit the "`" and the <type>s from the <lambda expr>s that follow behind the ":" of the <for expr>. Suppose v and w are two linear arrays of reals with equal descriptors, and interpret them as vectors. Then their sum-vector is given by

```
for v || w : ( - , (vi,wi) ) -> vi+wi rof
```

and their inner-product by

```
fold (real_add,0.) for v || w : ( - , (vi,wi) ) -> vi*wi rof
```

Next suppose a is a linear array of integers, we can multiply each entry by its position number by forming

```
for a : (i,ai) -> i*ai rof      { or shorter: for a : int_mul rof }
```

or form a matrix of all products of two entries by

```
for a , a : ( (-,-) , (ai,aj) ) -> ai*aj rof
```

Now assume m and n are two real matrices, then it will follow from the sequel that 'm<[ ][ ]>' is m viewed as row of rows, while 'n<[2][1]>' is n viewed as row of columns. We may now select the diagonal of m (if it is a square matrix) by

for m<[ ][ ]> : (i,mi_) -> mi_[i] rof

Also the matrix-prodoct m*n can be formed as follows, using an innerproduct function in_prod:

for m<[ ][ ]> , n<[2][1]> : ((-,-) , (mi_,n_j)) -> in_prod(mi_,n_j) rof

II.2.6.3 Descriptor Transformations

The <for expr> provides a powerful tool for combining arrays, nevertheless its handling of descriptors is rather limited. Therefore another class of expressions is added, the <descriptor transformation>s, which perform little computation, but rearrange components of arrays into new arrays, to make them suited for the other operations.

<descriptor transformation>
    ::= <primary> , <modifier> .

The type rule depends on the <modifier>.
strength(<primary>) = strength(<descriptor transformation>).

In any case for a <descriptor transformation> to reduce it is required that the <primary> be reduced (to an array) first.

<modifier>
    ::= <permuter> | <trimmer> | <slicer> | <paster> .

Permuters - The index positions may be permuted (the prime example is matrix transposition) using a <permuter> specifying the permutation with integral numbers:

<permuter>
    ::= "<[" , <integer denotation>LIST , "]>" .

Let n be the number of <integer denotation>s;
it is required that they form a permutation of the set {1,..,n}.
type(<descriptor transformation>) = type(<primary>) = [n]t (for some type t).

A <descriptor transformation> of the form 'A P' where P is a <permuter>, reduces to an array R defined as follows: for 1<=i<=n, let the i-th <integer denotation> in P be the number 'p(i)', and let A have a descriptor (b1,..,bn), then R has descriptor (bp(1),..,bp(n)), and the component of R at index 'ip(1),..,ip(n)' equals the component of A at the index 'i1,..,in'.

As an example, assume m is a matrix, then 'm<[2,1]>' denotes the transpose matrix.

Trimmers - The correspondence of indices to components of the array can be modified on a "per index position" basis by <trimmers>, the options being "reverse" (~), "lift lower bound to n" (; n), "lower upper bound to n" (: n), "shift to get lower bound at n" (at n):

<trimmer>
    ::= "<[" , <trim>LIST , "]>" .
<trim>
    ::= "~" | (";" | ":" | "at") , <expression> | <empty> .

Let n be the number of <trim>s in the LIST;
type(<descriptor transformation>) = type(<primary>) = [n]t (for some type t)
type(<expression>) = int,

strength(<expression>) = strong.

A <descriptor transformation> of the form 'A X', where X=<[X1,..,Xn]> is a <trimmer> reduces to an array R defined as follows: let A have descriptor ((l1,u1),..,(ln,un)) the R has a descriptor ((l'1,u'1),..,(l'n,u'n)), and the componont of R at the index 'j1,..,jn' equals that of A at 'k1,..,kn', where the l'i,u'i and ki are defined depending on the <trim> Xi, according to the following tabel:

Xi	l'i	u'i	ki
<empty>	li	ui	ji
~	li	ui	li+ui-ji
; n	max(n,li)	ui	ji
: n	li	min(n,ui)	ji
at n	n	ui+n-li	ji+li-n

We list a few examples using linear arrays:

[[4,2,6,7]] <[~]> = [[7,6,2,4]]
[[4,2,6,7]] <[:2]> = [[4,2]]
[[4,2,6,7]] <[;2]> = [[2,6,7]] <[at2]>
[[4,2,6,7]] <[;2]> <[:3]> <[~]> <[at1]> = [[6,2]]

We note that the last example may be written '[[4,2,6,7]] <[;2:3~at1]>' in the full language.

Slicers - An array may be "chopped up" into an array of sub-arrays by a <slicer>:

<slicer>
::= "<[" , <empty>LIST1 , "][" , <empty>LIST2 , "]>" .

Let m be the number of <empty>s in LIST1, n the number in LIST2;
type(<descriptor transformation>) = [m][n]t,
type(<primary>) = [m+n]t
(for some type t).

A <descriptor transformation> 'A S' where S is a <slicer> and m and n are as above, reduces to an array R defined as follows: let the descriptor of A be (b1,..,bm,bm+1,..,bm+n), then the descriptor D of R is (b1,..,bm) and its component at the index 'i1,..,im' within D is an array with descriptor (bm+1,..,bm+n), and whose component at index 'j1,..,jn' equals the component of A at 'i1,..,im,j1,..,jn'.

As an example, a matrix m may be viewed as a row of rows by writing 'm<[][]>'.

Pasters - An inverse operation to the <slicer> is given by the <paster>:

<paster>
::= "<[" , <empty>LIST1 , "|" , <empty>LIST2 , "]>" .

Let m be the number of <empty>s in LIST1, n the number in LIST2;
type(<descriptor transformation>) = [m+n]t,
type(<primary>) = [m][n]t,

(for some type t).

A <descriptor transformation> 'A P' where P is a <paster> and m and n are as above, reduces to an array R defined as follows: let the descriptor of A be (b1,..,bm) it is required that all components of A are arrays with equal descriptors (bm+1,..,bm+n) (otherwise 'A P' reduces to error; in case A has no components at all, we put bi=(1,0) for m+1<=i<=m+n), then the descriptor D of R is (b1,..,bm,bm+1,..,bm+n), and its component at an index 'i1,..,im,j1,..,jn' within D equals the component of the component of A at 'i1,..,im' at 'j1,..,jn'.

So e.g we have for any matrix m:

```
m <[ ][ ]> <[ | ]>     =     m
```

and conversely for any row-of-rows r for which 'r <[|]>' isn't "error":

```
r <[ | ]> <[ ][ ]>     =     r
```

This completes the <descriptor transformation>s.

II.2.6.4 Array Updates

It is suspected that, in spite of the power of these operations on arrays, programmers will have need for the flexibility they are used to in imperative languages (where you can go about updating an array randomly). Therefore we provide <array update>s which come in two kinds:

```
<array update>
    ::= <component update> | <exchange> .
```

Component Updates - The <component update> adjusts one component of an array, leaving the others unchanged.

```
<component update>
    ::= <primary> , "([" , <expression>iLIST , "]:=" , <expression>O , ")" .

Let n be the number of <expression>s in the LIST;
type(<component update>) = type(<primary>) = [n]t,
type(<expression>O) = t
(for some type t),
type(<expression>i) = int,
strength(<primary>) = strength(<expression>O) = strength(<component update>),
strength(<expression>i) = strong
```

A <component update> 'A ([i1,..,in]:= C)' reduces to an array R defined as follows: it is required that A and C be reduced first, as well as all the ij (1<=j<=n); the descriptor D of R equals that of A, the component of R at the index 'i1,..,in' equals C, the component of R at any other index equals that of A at the same index.

As an example we have '[[4,2,6,7]] ([4]:=5) = [[4,2,6,5]]'.

Exchanges - The <exchange> echanges components or sub-arrays of an array.

```
<exchange>
    ::= <primary> , "([" , (<empty>|<expression>)iLIST1
                  , "]<->[" , (<empty>|<expression>)iLIST2
                  , "])" .
```

It is required that both LISTs contain equally many items, say n,
and that the items at the i-th position in LIST1 and LIST2
be either both <empty> or both an <expression> (1<=i<=n);
type(<exchange>) = type(<primary>) = [n]t (for some type t),
type(<expression>i) = int,
strength(<primary>) = strength(<exchange>),
strength(<expression>i) = strong.

An <exchange> 'A ([i1,..,in]<->[j1,..,jn])' reduces to an array R defined as follows: let S be the set of those integers k for which ik isn't <empty>; it is required that A be reduced first, as well as the ik and jk for all k in S, and each ik and jk should lie in the interval specified by the corresponding bound-pair bk in the descriptor of A (otherwise the <exchange> reduces to "error"); the descriptor D of R equals that of A; let 'p1,..,pn' be an index within D such that pk = ik (respectively pk = jk) for all k in S and let 'q1,..,qn' be the index within D such that qk = jk (respectively qk = ik) for all k in S and qk = pk for all k not in S, (note that if both situations mentioned occur for the same index 'p1,..,pn', then both variants give the same index 'q1,..,qn', so there is no ambiguity) then the component of R at the index 'p1,..,pn' equals the component of A at the index 'q1,..,qn'; the component of R at any other index equals that of A at the same index.

As examples we have '[[4,2,6,7]] ([4]<->[2]) = [[4,7,6,2]]', and for a matrix m we have that 'm([1,]<->[3,])' is that matrix with the first and third row interchanged, while 'm([,3]<->[,4])' has the third and fourth column interchanged.

II.2.6.5 Operations for Linear Arrays

Finally, there are some operations that are defined only for arrays of dimension 1 (they may be useful for other arrays via <slicer>s and <paster>s).

Array Displays - Array displays provide a means for specifying fixed-length 1-dimensional arrays by enumerating the components:

<array display>
    ::= "[[]]" | "[[" , <expression>iLIST , "]]" .

type("[[]]") = @t []t,
type(<array display>) = []type(<expression>i),
strength(<expression>i) = strength(<array display>).

The empty <array display> '[[]]' reduces to the array with descriptor (1,0) (and no components). Any other <array display> '[[E1,..,En]]' reduces to an array R defined as follows: it is required that all Ei are reduced first (1<=i<=n); R has a descriptor D=(1,n) and for i within D the component of R at i equals Ei.

Built-in Functions - There is a number of built-in functions for handling linear arrays. We describe them by giving declarations for them (in the full language, for reasons of readability), expressing them in terms of already defined operations; we assume however that they can be implemented more efficiently than their defining counterparts.

```
let @t ( ([]t->int) -> ( []t -> ([]t,[]t) ) )
    split { function that splits a lineair array into two parts }
    = %t `locate { function determining the index to split the array at }
        -> `a { the array to be split }
            -> let m = locate a in ( a<[:m]> , a<[;m+1]> )
```

```
,   @t ( ([]t,[]t) -> []t )
    concatenate { function glueing together two linear arrays }
    = %t `(a1,a2) { the pair of arrays }
        -> let (l1,u1) = descr a1 , (l2,u2) = descr a2
           in tab (l1,u1+u2+1-l2)
              :   i -> if i<=u1 then a1 else a2<[at u1+1]> fi [i]
              bat

;   @t @s ( ( ((t,s)->s) , s ) -> ( []t -> s ) )
    fold    { a function combining the elements of a lineair array
              by some operation (e.g. summation) into one value }
    = %t%s `(operation,start) { combining operation and start value }
          -> rec f :
             `a { array to be "folded" }
             -> let (l,u) = descr a
                in if l>u then start
                   else operation( a[l] , f(a<[;l+1]>) )
                   fi

,   @t @s ( (s->(*|(t,s))) -> ( s -> []t ) )
    cumulate { function creating a linear array by repeatedly applying a
               "generator" function to a start value, the result being
               either 'stop' indicating termination of the process, or
               'include(e,n)', indicating that e is an element to be included
               into the array, and that n is the new start value.
               Note that the constructors 'stop' and 'include' are declared in
               the initial environment specially for the function cumulate }
    = %t%s `f { the generator function }
          -> rec cumulate_from :
             `start { the initial value }
             -> case f start { inspect first application of f to start }
                of (stop) -> [[]]$t { yield an empty row of type []t }
                |  (include(elem,new_start))
                   -> concatenate( [[elem]] , cumulate_from new_start )
                esac

;   @t ( (t->bool) -> ([]t->[]t) )
    select { function that selects those components from a linear array,
             that satify a given predicate, and collects them into an array }
    = %t `predicate
        -> cumulate
           ( rec ([]t->(*|(t,[]t))) search :
                 { look for first component satisfying predicate }
             `start { array to search in }
              -> let (l,u)=descr start ; first=start[l] , rest=start<[;l+1]]>
                 in if l>u then stop
                    elif predicate first then include(first,rest)
                    else search rest
                    fi
           )
,   @t ( [](int,t) -> []t )
    random_write
    { a function that takes an array of (index,component)-pairs, and builds a new
      array with equal descriptor, and has at each index the indicated component.
      Imperatively this could be realised by starting with an empty row r, and for
      each pair (i,c) perform the assignment r[i]:=c. We require that each
      component is assigned to exactly once, i.e. the index-parts must form a
      permutation of the legal index values. Note that writing this function
      directly in a functional language invloves searching at each index value
      for a pair with that index-part, which is significantly less efficient than
      the imperative approach. That's why it's a built-in function }
```

```
= %t `a { the array with (index,component)-pairs }
     -> tab descr a
        : i -> (rec (int->t) search_for_i_from: `j ->
                  if j>upb a then error "random_write: no permutation"
                     { imperatively this error would be discovered
                       by a collission (an index present twice)
                       rather than by a search fail (an index not present) }
                  elif i = (1_of_2 a[j]) then 2_of_2 a[j]
                  else search_for_i_from(succ j)
                  fi
                 ) (lwb a)
     bat
in ...
```

We give one example of the use of each function:

```
split (` []int- -> 3) [[4,2,6,7]]    =    ( [[4,2,6]] , [[7]]<[at4]> )
concatenate [[4,2,6]] [[7]]          =    [[4,2,6,7]]
fold (int_add,0) [[4,2,6,7]]         =    19
cumulate$int$int
 (`n -> if n=0
        then stop
        else include(3*n,n-1)
        fi
 ) 5                                 =    [[15,12,9,6,3]]
select (`int n-> n>3 ) [[4,2,6,7]]   =    [[4,6,7]]
random_write [[(2,6),(3,7),(1,4)]]   =    [[4,6,7]]
```

This (finally) finishes the section on arrays, and the description of the core language, with exception of the remaining built-in functions. These are described in section II.4 on the initial environment.

II.3 EXTENSIONS

We now give the extensions to the core language that give the full language. We will give the extensions in several sections, that each have the purpose of making a particular construct available. Some extensions necessarily follow other ones, as they modify a construct not present in the core language. Also some extensions may form a construct that may be subject to the same extension, in a repetitive (if you like: recursive) fashion.

There are two ways in which one can view extensions:

1. as particular combinations of elements of the core-, or less extended language, for which special syntactic constructs are created,
2. as part of the full language, for which the semantics are defined by giving an equivalent in a less extended form of the language.

Which of the views you prefer depends on whether you like going from semantics to syntax, or rather from syntax to semantics. We take the first point of view, which is the more natural one when designing a language.

The form in which we will present the extension rules, is as follows. First we give a pattern describing a construct in the restricted language, using BNF-style expressions. Then similarly we give a corresponding construct in the extended language, that is equivalent to the former construct. Doing so, there will of course have to be a correspondence between the syntactic elements (at least the non-terminal ones) that appear in both constructs. So we will make the convention that any entity that appears more than once, will denote the same terminal production in all occurrences, except when they are followed by different digits or letters (rather similar to the description of the core language, where we talked about type(<expression>) and the like, indicating the occurence of <expression> in

the syntax rule; but <expression>0 is distinguished from <expression>i etc.). The part of the extension rules that defines the extended construct, actually defines new syntax, so it may introduce new syntactic categories, as well as new production rules for them or existing categories. Therefore, we do not only give the final form of the extension, but may also give "approximating" expressions for that final form, indicating that there are assumed to be syntactic rules, that allow the final form to be produced from the approximating ones. For reasons of clarity we will also give approximations of the pattern for the unextended construct, making visible its grammatical structure, and the syntactic category from which it is produced. Naturally, the extended construct is a (new) production of that same category. The format in which we present the extension will be:

-> <approx>1 ; <approx>2 ; .. ; <approx>n : <unextended construct>
<- <approx>'1 ; <approx>'2 ; .. ; <approx>'m : <extended construct> .

So by what is said above, <approx>1 is a single syntactical category, and '<approx>1 => <approx>2 => .. => <approx>n => <unextended construct>' should incorporate some series of legal productions of the restricted syntax (it may be even more permissive at certain points, when the extension is recursive, enabling match to an extended construct) while the syntax should be extended so that '<approx>1 => <approx>'1 => .. => <approx>'m => <extended construct>' becomes a series of legal productions of the extended syntax.

This format of definition is not always sufficient to describe extensions precisely, so we will describe some extensions wholely or partly by English text; mostly this will concern restrictions to the applicability of the rule given. On the whole our decription of extensions will be more informal than the description of the core language.

Though the extensions are given as lexical rewrite rules, they should be interpreted as transformations of the syntax tree: we don't want parts of the extended construct to parse differently from their occurrences the original. At the level of syntactic precedence, this may be guaranteed by requiring the original to be parenthesised in case there is any danger (we have tried to be precise in this respect), but a problem arises when a subexpression comes into a position in the extension of strength differing from its original strength. We have no means of forcing a certain expression to be in strong position, but luckily we can always transform an expression valid in a strong context to one that is valid in a weak context, by adding any required <type>s (the other way round is trivial: no alteration is needed at all). So if a subexpression that was in weak position, comes into strong position in the extended construct, any <type>s that become optional due to this increased strength may indeed be omitted, and are to be restored upon undoing of the extension (e.g. by the compiler). In case of a strength change in the reverse direction, we require that any subexpression coming into weak position by the extension should already be adorned with sufficient <type>s to meet its new position for the extension to be allowed. We will always note these strength changes explicitly; unless mentioned strengths are unchanged.

II.3.1 Miscellaneous Extensions

We start with giving some miscellaneous extensions, not particularly related to other ones, that may also serve as an illustration of the method of describing extensions.

II.3.1.1 Conditionals

Firstly, we introduce the well-known conditional expression as a a special case of the <case-of expr>, which is consistent with the choice of using the type '(*|*)' as Boolean.

```
-> <expression> ; <case-of expr>
;  "case",<expression>0,"of",<lambda expr>,"|",<lambda expr>,"esac"
:  "case",<expression>0,"of`*()->",<expression>1,"|`*()->",<expression>2,"esac"
<- <conditional>
;  "if",<expression>0,"then",<expression>1,<else part>
:  "if",<expression>0,"then",<expression>1,"else",<expression>2,"fi" .
```

So for example we have that 'if p then q else r fi'
stands for 'case p of ` *() -> q else ` *() -> r esac'.

We may extend for nested <conditional>s, like in Algol 68:

```
-> <else part>
;  "else",<conditional>,"fi"
:  "else if",<expression>0,"then",<expression>1,"else",<expression>2,"fi fi"
<- "elif",<expression>0,"then",<expression>1,<else part>0
:  "elif",<expression>0,"then",<expression>1,"else",<expression>2,"fi".
```

Note that, provided it is of the appropriate form, <else part>0 may be further extended by the same extension, and an arbitrarily deeply nested collection of <conditional>s may be reshaped into one.

II.3.1.2 Strings

We provide the usual notation for strings, enclosed in double-quotes:

```
-> <array display> ; "[[",<character denotation>LIST,"]]"
:  "[[",("'",(<any ascii character but quote>|<quote>)i)LIST,"]]"
<- <string>
:  <quote>,(<any ascii character but quote>|<quote image>)i+,<quote>.
```

Here '<quote image> :== <quote>,<quote>'. So <quote>s are doubled inside <string>s. The empty string needs a slightly different extension, because of the polymorhic type of the empty <array display>:

```
-> <enclosed expr> ; "(",<specialisation>,")"
;  "(",<array display>,"$char)"
:  "([[]]$char)"
<- <array display> ; <string> : <quote>,<quote> .
```

So, as an example (for one time omitting our meta-delimiters '')
"""This"" is a 'string'."
stands for
[['","T,'h,'i,'s,'",' ,'i,'s,' ,'a,' ,'','s,'t,'r,'i,'n,'g,'','.]]
and
"This" is a 'string'.
is the way it will be printed.

II.3.1.3 Subscriptions

Next we give a shorthand for <subscription>s in which an out-of-bounds index should result in "error", which is expected to be a quite common situation.

```
-> <subscription> ; <primary>,"[",<expression>LIST,"ext",<lambda expr>,"]"
:  <primary> ,"[",<expression>LIST
             ,"ext",`<type>0,"- ->"
                  ,"error",<quote>,"Subscript out of bounds",<quote>,"$",<type>1
             ,"]"
<- <primary>,"[",<expression>LIST,"]".
```

So 'a[i,j]' where a is of type '[,]t', stands for
'a[i,j ext `(int,int)- -> error "Subscript out of bounds"$t]'.

II.3.1.4 Descriptor Transformation Merging

The core language defines several kinds of <decriptor transformation>s, nested forms of which may be merged to a more compact form.

To begin with, several <trimmer>s may be combined into one:

```
-> <decriptor transformation> ; <decriptor transformation>1,<trimmer>
;  <primary>,<trimmer>,<trimmer>
:  <primary>,"<[",<trim>*iLIST,"]> <[",<trim>iLIST,"]>"
<- <primary>,<trimmer>
;  <primary>,"<[",(<trim>+)LIST,"]>"
:  <primary>,"<[",(<trim>*i,<trim>i)LIST,"]>".
```

So <trim>s are just juxtaposed at the proper index position, in their original order. So we have e.g. that 'a<[;2at7:15~]>' stands for 'a<[;2]><[at7]><[:15]><[~]>'. It is easily seen that the slight ambiguity due to the possibiltity of a <trim> being <empty> is semantically neutral.

We allow a sequence of successive <slicer>s to be merged into one.

```
-> <descriptor transformation> ; <descriptor transformation>1,<slicer>
;  <primary>,<slicer>,<slicer>
:  <primary>,"<[",<empty>LISTO,"][",<empty>LIST1,"]>"
           ,"<[",(<empty>LISTi)+CHAIN"][", "]>"
<- <primary>,<slicer>
;  <primary>,"<[",(<empty>LIST)+CHAIN"][", "]>"
:  <primary>,"<[",(<empty>LISTi)+CHAIN"][", "][",<empty>LIST1,"]>".
```

Note that, by type constraints, '<empty>LISTO' contains the same number of <empty>s as '(<empty>LISTi)CHAIN"]["'. We have e.g. that 'a<[ ][,][ ]>' stands for 'a<[, ,][ ]><[ ][,]>'. As a result of this extension we have '<slicer> ::= "<[",(<empty>LIST)+CHAIN"][", "]>"'.

Similarly, we have for <paster>s:

```
-> <descriptor transformation> ; <descriptor transformation>1,<paster>
;  <primary>,<paster>,<paster>
:  <primary>,"<[",(<empty>LISTi)+CHAIN"|", "]>"
           ,"<[",<empty>LISTO,"|",<empty>LIST1,"]>"
<- <primary>,<paster>
;  <primary>,"<[",(<empty>LIST)+CHAIN"|", "]>"
:  <primary>,"<[",(<empty>LISTi)+CHAIN"|", "|",<empty>LIST1,"]>".
```

Again, '<empty>LISTO' contains the same number of <empty>s as '(<empty>LISTi)CHAIN"|"'. We have e.g. that 'a<[| , |]>' stands for 'a<[| ,]><[, , |]>'. As a result of this extension we have '<paster> ::= "<[",(<empty>LIST)+CHAIN"|", "]>"'.

We also allow a combination of a <permuter>, <trimmer> and <slicer>, or subset thereof, in that order, to be merged into one. The separators ("," or "][") are taken from the <slicer>, the items of the <permuter> and the <trimmer> are juxtaposed in between. If no <trimmer> or no <slicer> actually participates in the merging, we put <null> in its place, with '<null> ::= "<[",<empty>LIST,"]>"'; actually <null> is a <trimmer> with trivial semantics. It is easily seen that the productions of '<slicer>|<null>' could equivalently be described by '"<[",<empty>CHAIN("," | "]["),"]>"'.

```
-> <decriptor transformation>
;  <primary>,<permuter>?,<trimmer>,(<slicer>|<null>)
:  <primary>,("<[",<integer denotation>iLIST,"]>")?
           ,"<[",<trim>*iLIST,"]>"
           ,"<[",<empty>CHAIN((","  |  "]["")j),"]>"
<- <primary>,<merged modifier>
:  <primary>,"<[",(<integer denotation>i?,<trim>*i)CHAIN((","  |  "]["")j),"]>".
```

So e.g. 'a<[3 , 1;1:10][2at7]>' stands for
'a <[3,1,2]> <[, ;1:10 , at7]> <[,][ ]>'.

II.3.2 Simple Declarations

Declarations are a very important extensions to the language, they allow identifiers to be bound to fixed expressions. The way to achieve this in the core-language is to introduce the <variable>(s) in a <lambda expr> and to apply this directly to the desired expression(s). The declarations are grouped in a <let expr> or a <where expr>.

```
-> <expression> ; <function application>
;  "(",<lambda expr>,")",<expression>0
:  "( `",<type>,<var plan>,"->",<expression>1,")",<expression>0
<- <let expr> ; "let",<declarations>,"in",<expression>1
;  "let",<simple declaration>,"in",<expression>1
;  "let",<formal>,"=",<expression>0,"in",<expression>1
:  "let",<type>,<var plan>,"=",<expression>0,"in",<expression>1.
```

So, schematically, 'let P=A in E' stands for '(`P->E) A'. There is a strength change involved in this extension. The strength of <expression>1 is weak in the unextended construct, but in the <let expr> its strength is equal to that of the <let expr> (since their types must also be equal).

We may separate the <simple declaration> in a <let expr> into a list of <simple declaration>s, if its structure allows this:

```
-> <declarations> ; <simple declaration>
;  <tuple type>,<compound plan>,"=",<tuple display>
:  "(",<type>i+LIST,"){",<var plan>i+LIST,")=(",<expression>i+LIST,")"
<- <simple declaration>+LIST
;  (<formal>i,"=",<expression>i)+LIST
:  (<type>i,<var plan>i,"=",<expression>i)+LIST
```

Schematically, 'T1 x1=a1 , .. , Tn xn=an' stands for '(T1,..,Tn)(x1,..,xn)=(a1,..an)'. Note that we have '<declarations> ::= <simple declaration>LIST' by now, the cases that LIST has one or more than one item being introduced separately; there will be other cases for <declarations> involving recursive declarations.

Some people like to have the <declarations> after the use of the declared identifiers, so we allow <declarations> to follow an expression in a <where expr>:

```
-> <tertiary>
;  "(",<let expr>,")" : "(let",<declarations>,"in",<tertiary>0,")"
<- <where expr> : <tertiary>0,"where",<declarations>,"end".
```

Clearly, allowing identifiers to be declared after their use makes any simple one-pass compiler impossible, while that might have been feasible for the language described upto this point. This may seem to be a high price for contenting those fond of define-after-use. However, if we want to allow sets of mutually recursive declarations (which we do), then this price will have to be paid anyway. Nevertheless, we emphasise that all context-free analysis, and part of the

context-sensitive analysis can be performed on the first pass over the program text. The latter category includes resolution of all ambiguities in the context-free grammar. Issues remaining for a later pass are mainly static identification and type-checking.

In one <let expr> the <simple declaration>s are separate, and their order is irrelevant. Often, however, new declarations depend on older ones, and to achieve this one has to use nested <let expr>s (or nested <where expr>s, but that really is going to look upside-down). The following extension allows "in let" to be contracted to ";":

```
-> <let expr> ; "let",<declarations>0,"in",<let expr>0
:  "let",<declarations>0,"in let",<declarations>iCHAIN";", "in",<expression>
<- "let", <declarations>+CHAIN";", "in",<expression>
:  "let", <declarations>0,";",<declarations>iCHAIN";", "in",<expression>.
```

Until now, the <formal>s occurring in <simple declaration>s must contain a <type>, which is a consequence of the <expression> following it being strong, like it was in the <function application> it was extended from, where that <expression> was in the (strong) argument position. Now we may omit the <type> in the <formal> provided we weaken the strength of the <expression>:

```
-> <simple declaration> ; <formal>0,"=",<expression>
; <typed formal>,"=",<expression>
:  <type>,<var plan>,"=",<expression>
<- <formal>1,"=",<expression>
:  <var plan>,"=",<expression>.
```

In the extended construct, strength(<expression>) = weak; the omitted <type> can now be reconstructed to be type(<expression>). So 'let x=3 in x+x' is valid, and stands for 'let int x=3 in x+x', but on the other hand you may not write 'let f=`x.x+x in f 3', since the untyped <lambda expr> may not stand in a weak position. Note that you could not tell in this last example whether f should perform integer or real (or some other) addition, without searching for applied occurrences of f. The type-requirements of TALE have been designed to allow type-checking without any global searching or equation-solving. All type dependencies form a partial ordering (no cycles), so that types can be derived and checked in a straightforward way. Apart from being convenient for the compiler and human reader, this property also enables the incorporation of overloaded opertors (like '+' denoting both integer and real addition) into the language (see section II.3.9). This would be almost impossible in the presence of cyclic type dependencies.

For <typed formal>s we allow the <type> to be interleaved with the <var plan> in case of tuples:

```
-> <typed formal> ; <tuple type>,<compound plan>
:  "(",<type>i+LIST,")(",<var plan>i+LIST,")"
<- "(",<typed formal>i+LIST,")"
:  "(",(<type>i,<var plan>i)+LIST,")".

-> <typed formal> ; <tuple type>,<compound plan> : "*()"
<- "()".
```

So e.g. '(int x,real y)' stands for '(int,real)(x,y)'.

II.3.3 Mutual Recursion and Recursive Declarations

One might wonder why, after having defined the substitution symbol 'E[P:=A]' for an arbitrary <var plan> P, it is used in <recursion expr>s only for the simple case of a <variable>. The answer is simple: if we had allowed a general <var plan> in a <recursion expr>, and had given the same simple substitution rule as semantics, this

would always lead to non-termination unless the <var plan> were a <variable> (or '-' which is useless). Indeed for the <recursion expr> R to reduce, we would require R to be reduced to a tuple first; a circular demand. Nevertheless there is a valid interpretation for a <recursion expr> with a <compound plan>, but it requires more subtle semantics: instead of demanding R to reduce to a tuple, we should form <function application>s, of "projection functions" applied to R, and these should be substituted for the <variable>s in the <var plan>. Reduction of R can then proceed, hopefully allowing R to be reduced to a tuple, and when need comes, the projections of R may be taken. In this way we may specify mutually recursive functions or other objects. Note that given a <formal> containing a <variable>, we may write the projection function as ' `<formal> -> <applied var> ', where the <applied var> identifies the occurrence of <variable> in <formal>. We treat mutual recursion as an extension because it is easier to formulate it this way, than to give the right semantics directly. Also it shows that simple recursion is in principle sufficient.

```
-> <recursion expr> ; "rec",<type>?,<variable>0,":",<let expr>
;  "rec",<type>?,<variable>0,":"
   ,"let", (<variable>i,"=",<function application>)+LIST,"in",<expression>
:  "rec",<type>?,<variable>0,":"
   ,"let", (<variable>i
            ,"= (`",<type>,<var plan>,"->",<applied var>i,")",<applied var>0
            )+LIST
   ,"in",<expression>
<- "rec",<formal>,":",<expression>
:  "rec",<type>?,<var plan>,":",<expression>.
```

The following points should be noted:

- all ocurrences of <var plan> are identical, idem for <applied var>0,
- <variable>0 does not have any applied occurrences within <expression>,
- every <applied var>0 identifies <variable>0
- the <variable>i run through the set of <variable>s in <formal>, from left to right,
- we haven't renamed variables, so <applied var>i has the same textual representation as <variable>i, but it identifies the ocurrence of <variable>i in <formal>, and not the <variable>i preceding the "=" (it's not even in its scope).
- if the optional '<type>?' is absent, <type> is the unique type that would be valid at the place of the '<type>?' (determined by the context).

We give one simple example:

'rec (t1,t2) (x,y) : p x y' stands for

```
'rec (t1,t2) z :
     let x = ( `(t1,t2)(x,y) -> x) z
     ,   y = ( `(t1,t2)(x,y) -> y) z
     in p x y
'
```

(imagine p is of the form `x->`y->(A,B); check that this reduces properly).

We now allow this mutual recursion to be specified in recursive declarations, which are part of a distinguished (by "rec") kind of <let expr>s:

```
-> <let expr>
:  "let",<typed formal>,"= rec",<typed formal>,":"<expression>0
   ,"in",<expression>1
<- "let",<declarations>,"in",<expression>1
;  "let rec",<recursive declaration>,"in",<expression>1
:  "let rec",<typed formal>,"=",<expression>0,"in",<expression>1.
```

So, e.g. 'let rec t f=F in E' stands for 'let t f=rec t f:F in E'. Note that two identical <typed formal>s are replaced by one in the extension; consequently the latter ranges over both <expression>0 and <expression>1.

We allow the same separation of declarations as in the <let expr>:

```
-> <declarations> ; "rec",<recursive declaration>
;  "rec",<tuple type>,<compound plan>,"=",<tuple display>
:  "rec (",<type>i+LIST,")(",<var plan>i+LIST,")=(",<expression>i+LIST,")"
<- "rec",<recursive declaration>+LIST
;  "rec",(<typed formal>i,"=",<expression>i)+LIST
:  "rec",(<type>i,<var plan>i,"=",<expression>i)+LIST .
```

Note that we have:

```
<declarations> ::= <simple declaration>LIST
                 | rec",<recursive declaration>LIST .
```

by now.
This automatically allows <where expr>s to be recursive and nested <let expr>s to be combined with "; rec" instead of "in let rec". However, we do not allow that the <formal> in a <recursive declaration> is changed from a <typed formal> to a <var plan>, as was allowed in <simple declaration>s. The reason for this is, that in order to determine the type of the <expression> we need to know the type of any <applied var>s in it, and that may depend on the type of the <formal> of that same <recursive declaration>. If the type of that <formal> would have to be derived from the <expression>, then we would have a cyclic type dependency. The necessary specification of a type in <recursive declaration>s is the only syntactical difference with <simple declaration>s. When we talk about declarations in general, we will use <declaration> to indicate either of them (though <declaration> doesn't occur in the official syntax).

II.3.4 Type Declarations

In the core language, any <type> must be spelled out in full. This involves no efficiency penalty, because <type>s don't get into terms, but it can be rather tedious if some rather large <type>s are used often. Therefore we give an abbreviation system for <type>s. We emphasize that it is no more than that, and no new types are involved. As a shorthand for some more complicated <type>, we will use <type var>s. This is a logical choice, but there may some confusion, as there are already three uses for <type var>s in the core language:

- bound by '%', to be used as primitive type in the following <expression>,
- bound by '@', to be used within a type to indicate polymorphism,
- bound by "rectype" to indicate a cyclic type structure.

In neither of these cases is the <type var> a shorthand for something else. We introduce yet two more uses of <type var>s, both of which are shorthands.

First we introduce a local abbreviation within a type:

```
-> <type>0
<- <type generator>,("$",<type>i)+
:  ("%",<type var>i)+,<type>1,("$",<type>i)+.
```

Some explanation is required here. A <type generator> acts as a function producing types from types, The '<type>i's that follow it act as argument types. The <type generator> itself is specified by giving a sequence of "formal parameters" in the form of <type var>s (preceded by '%' acting as "lambda symbol" for types), that bind free occurrences of these <type var>s inside <type>1. Because a <type generator> is not in itself a type, and there is no use for them in the core language, they must occur fully instantiated, i.e. there must be as many <type>s following it, as there are binding <type var>s inside it. The correspondence between extended and unextended form is that <type>0 is obtained by substituting every <type>i for the corresponding <type var>i into <type>1. So, e.g. '%t(t,t,t)$int'

stands for '(int,int,int)'. We call the types that can thus be formed, 'substitution instances' of the <type generator>.

This extension may reduce the amount of text needed to specify a type, but that is not the main reason for giving it. We want to have type declarations to be used within expressions, and we want to be able to provide a name, not only to single types, but also for families of types like "list", that need one or more parameter types to indicate a actual type. Therefore the category <type generator> was introduced. We now present the type declarations.

```
-> <expression>0
<- <type declaration expr>
;  "type",<type declaration>,"in",<expression>1
:  "type",<type var>,"=",(<type>|<type generator>),"in",<expression>1.
```

The correspondence is that <expression>0 is obtained by substituting the <type> or <type generator> for <type var> into <expression>1. In case a <type var> is thus declared to stand for a <type generator>, the term "type var" and the keyword "type" are somewhat misnomers.

We give an example. If we declare 'type pair = %t(t,t) in E', then inside E, 'pair$char' will stand for '%t(t,t)$char', which in turn stands for '(char,char)'.

II.3.5 Forgetful Extensions

We define a number of extensions, whose sole purpose it is to allow a reduction of the number of symbols required to specify a program when this can be achieved without ambiguity.

II.3.5.1 Dropping Lambdas

The first of these forgetful extensions stems from the lambda-calculus: "repeated lambdas" may be contracted:

```
-> <lambda expr> ; "`",<formal>+0,"->",<lambda expr>1
:  "`",<formal>+0,"-> `",<formal>,("."|"->"),<expression>
<- "`",<formal>+1,("."|"->"),<expression>
:  "`",<formal>+0,<formal>,("."|"->"),<expression>.
```

So, e.g. ''x (y,z) -> E' stands for ''x -> `(y,z) -> E'.

In the core language, <limb>s were introduced wherever a subexpression may need access to an intermediate value (or expression), that is produced by the semantics of the surrounding construct, as result of some operation performed upon other subexpressions. This access was realised in the semantics by producing a <function application> of the <limb> to that value. Since it is not very likely that the <limb> will achieve its effect simply by applying an already declared function to that value, it will most probably accept the value by binding it to a <formal>, i.e. the <limb> will be a <lambda expr>. It is however natural to think of the <formal> not as part of a <lambda expr> but just as a means for the <limb> to name the intermediate value. Also, a <lambda expr> may require a <type> to be present in the <formal>, while in a <limb> this type can always be deduced from the context. Therefore we allow the "`" and the <type> to be omitted in <limb>s:

```
-> <limb> ; <lambda expr> : "`",<type>,<var plan>,"->",<expression>
<- <var plan>,"->",<expression>.
```

So we may shorten 'tab(1,10): `int n -> n*n bat' to 'tab(1,10): n -> n*n bat.'

II.3.5.2 Lambda-Case Expressions

It will often occur that a function performs case-analysis upon its parameter, thereby introducing names for (parts of) the component in each variant separately (possibly using the extension above), and so the <variable> introduced in the <lambda expr> is used only once. We give an extension allowing to remove the <variable> altogether in such cases:

```
-> <expression> ; <lambda expr> ; "`",<type>?,<variable>,"->",<case-of expr>
:  "`",<type>?,<variable>,"-> case",<applied var>,"of",<caselimbs>,"esac"
<- <lambda-case-of expr>
:  "case",<type>?,"of",<caselimbs>,"esac".
```

Here of course <applied var> identifies <variable>, and there are no other applied occurrences of <variable>. So, schematically 'case of L1 |..| Ln esac' stands for '`x->case x of L1 |..| Ln esac'.

We may formulate a similar extension for <case-in expr>s:

```
-> <expression> ; <lambda expr> ; "`",<formal>,"->",<case-in expr>
:  "` int",<variable>,"->"
  ,"case",<applied var>,"in",<expression>iLIST,"out",<limb>,"esac"
<- <lambda-case-in expr>
:  "case in",<expression>iLIST,"out",<limb>,"esac".
```

where, as above, <applied var> identifies <variable>, and there are no other applied occurrences of <variable>. So, schematically 'case in E1 ,.., En out F esac' stands for '`int n->case n in E1 ,.., En out F esac'.

II.3.5.3 Omitting Specialisations

Another forgetful extension concerns <specialisation>s. In certain circumstances it is not necessary to specify the specialisation, namely when the required specialisation can be inferred from the context. Because we have required that a @-bound <type var> is always used at least once, and therefore there is at most one way to specialise to a given type, we may always deduce the required specialisation when the context is strong. When we do omit the specilisation, the same expression gets two types, namely the (polymorpic) one before specialisation (the a-priori type) and the one after specialisation (the a-posteriori type). The a-priori type appears in relations for types of its own subexpressions, while the a-posteriori type is used in "external" type relations. In fact this is similar to the situation with Algol-68 "coercions", except that there are no implied semantics (because there are none for <specialisation>s). We allow this extension to take place only on expressions that would not propagate strong contexts further inward, since otherwise there would be ambiguity as to at which level the coercion should be inserted. We therefore put:

```
<coercend>
    ::= <applied var> | <function application> | <formula> | "[[]]" | "(<>)" .
```

and we have the extension

```
-> <expression> ; <specialisation>
:  <coercend>,("$",<type>)+
<- <coercend>.
```

where we require that 'strength(<expression>)=strong'.

Neither of these two rules should be taken too formally, since <coercend> will not occur in the grammar of the language. The use of '<coercend>' merely indicates that, in order for the omission of the specialisation to be allowed, the expressions at the indicated positions should be one of the indicated forms (and this should not

be altered by further extensions). Also, it may be necessary to parenthesise either the extended or the unextended construct, as required by the precedence structure.

There is an important situation we missed with this extension, and that is where the <specialisation> is the <tertiary> (i.e. the function part) of a <function application>, which is always a weak position. As polymorphic objects are generally functions, and above position, though not fully fixing one required type, in any case needs a non-polymorphic (namely function-) type, we have a good chance of predicting the specialisation needed. In order to succeed, we need a mild extra condition, namely that the argument of the function is weak (so that we know its type) and that the type substituted occurs in the left (i.e. parameter-) part of the <function type> (a polymorhic function type, using its <type var> only in the result part is a bit strange anyway).

```
-> <function application> ; <specialisation>,<primary>
:  "(",<tertiary>,")",("$",<type>)+,<primary>
<- <tertiary>,<primary>.
```

We needn't restrict to the case of <coercend>s here, since the <tertiary> is weak, and no strength propagation occurs anyway. As said above, there is a strength change: 'strength(<primary>)=weak' in the extended construct (while it was strong before the extension). Also the a-priori type of the <tertiary> must be of the form '("@",<type var>)+,"(",<type>0,"->",<type>1,")"' with all of the <type var>s occurring in <type>0. Summarising: in a <function application> the <tertiary> is always weak; if it has a polymorphic type, then the <primary> is weak too, and the <tertiary> is implicitly specialised with types deduced from the type of the <primary>; otherwise there is no implicit specialisation and the <primary> is strong.

II.3.6 Constructors

In the core language, the <union display> is the only way to obtain expressions of union type, but it looks rather ugly (especially if the <type>s are present), and for frequently used union types one would like to have "injection functions" that would also provide a names for the different variants of the union type. These names could also clarify <case-of expr>s as a label of the limb chosen. We provide a mechanism for creating such names. They are indicated by identifiers, but they get a special status, enabling extensions to the <case-of expr> and are called "constructors". We allow constructors not only for particular union types, but also for families of types denoted by <type generator>s, in which case we get polymorphic injection functions.

II.3.6.1 Constructor Declarations

The simpler use of constructors is as injection functions, mapping a summand domain of a disjoint union into it. We make special provision for the case a component type of the union is '*', since then the corresponding injection function has but one value in its image, and we identify the constructor with that value instead of with the function.

To facilitate the notation of this extension, we temporarily introduce the notation 'INJECT-i-n<expression>' to indicate the <union display> of the form '"(", (<expression>|<empty>)+CHAIN"|" ,")"', where there are n items in the +CHAIN, the i-th of which is <expression>, all the others <empty>.

```
-> <expression> ; <let expr> ; "let",<simple declaration>+LIST,"in",<expression>0
;  "let",(<type>0i,<variable>i,"=",<expression>i)+LIST,"in",<expression>0
:  "let",( ("@",<type var>j)*
         , ( "(",<type>1i,"->",<union type>,")" | <union type> )
         , <variable>i,"="
         , ("%",<type var>j)*
         , ( "`",<variable>,"->",INJECT-i-n<applied var>,")" | INJECT-i-n"()" )
         )+LIST
  ,"in",<expression>0
<- <constructor declaration expr>
;  "constructors",<constructor declaration>,"in",<expression>0
;  "constructors",<variable>i+LIST,"for",(<type>|<type generator>)
  ,"in",<expression>0
;  "constructors",<variable>i+LIST,"for",("%",<type var>j)*,<union type>
  ,"in",<expression>0
:  "constructors",<variable>i+LIST
  ,"for",("%",<type var>j)*,"(",<type>1i+CHAIN"|",")"
  ,"in",<expression>0.
```

Note the following points

- i counts the variants of the union type, n is the total number of them
- j counts the parameter types of the <type generator>; if it happens to be a <type> instead, the corresponding items are repeated 0 times so j doesn't occur
- inside '<type>0i' and '<expression>i' there are two alternatives; which one is taken depends on whether '<type>1i' is '*': if so the second alternative is chosen, otherwise the first
- within '<expression>i', '<applied var>' identifies '<variable>'; its choice is arbitrary

We give an example of this rather complicated extension:

```
'constructors succeed,fail for %t(t|*) in E' is extended from

'let @t(t->(t|*)) succeed = %t `x->( x |     )
 ,   @t    (t|*)  fail    = %t     (   | () )
 in E
'.
```

II.3.6.2 Constructor Limbs

So above extension implies that an <applied var> that identifies a <variable> declared in a <constructor declaration>, simply indicates the corresponding injection function. However we now introduce another use of such identifiers, that is different from the use of ordinary variables. Having declared constructors for a type or family of types, it is natural to associate a variant of a union type to the corresponding constructor name. However, <case-of expr>s only distinguish the variants by their position number. We now allow this distinction to be made according to constructor names, and as a consequence allow the <limb>s of such a <case-of expr> to be permuted. Such an applied use of an identifier declared in a <constructor declaration> will be noted <constructor>; so '<constructor> :== <identifier>'.

```
-> <caselimbs> ; <limb>i+CHAIN"|"
:  ( <var plan>i,"->",<expression>i )+CHAIN"|"
<- <constructor limb>i+CHAIN"|"
;  ( <pattern>p(i),"->",<expression>p(i) )+CHAIN"|"
:  ( "(",<constructor>p(i),<var plan>p(i)?,") ->",<expression>p(i) )+CHAIN"|".
```

It is understood that the items <var plan>p(i) and <expression>p(i) are the same ones that appear in the unextended construct, but in permuted order, the permutation being noted 'p'. This permutation can be reconstructed from the <constructor>s: the <constructor> appearing in the i-th <constructor limb> identifies the p(i)-th <variable> in the <constructor declaration> in which it was declared. When the variant to which that constructor corresponds is '*' then the '<var plan>p(i)' is omitted in the extended construct (while it was '()' in the unextended construct), otherwise it is included. We require of course that the <constructor>s all be declared in the same <constructor declaration>, and that each such constructor be used exactly once. Also we require that 'type(<expression>0)' is the <type> appearing in that same <constructor declaration>, or in case a <type generator> appears there, a substitution instance of it. We specially note that <pattern>s are always parenthesised, and contain no commas at the level of these parantheses; this suffices to distinguish them from <var plan>s, that could also be legal in the same position.

In the scope of the <constructor declaration> of our previous example, we have that

'case u in (fail) -> E2 | (succeed x) -> E1 esac' stands for

'case u in x -> E1 | () -> E2 esac'.

II.3.7 Heuristic Application and Pattern Matching

In this section we give a number of extensions, that alter the appearence of certain constructs, slightly rearranging their constituents, to a form that some people find easier to understand, though its structure less resembles the semantics. A typical example (omitting types) is writing 'suc x=x+1' for the declaration 'suc=`x->x+1'. The reason people like the former form more, is that it declares suc to be bound to an algorithm (like in the latter form), but looks like it only states a fact about suc. We will fulfill aforementioned desire by extensions, but to a limited extent: you may write 'suc x=x+1' as above, but not 'x+1=suc x' nor 'suc (x-1)=x' nor even 'suc(x)=x+1'. So the programmer must be aware that the left-hand-side of such declarations only superficially resembles an <expression> like the right-hand-side, but obeys a quite different syntax, and has entirely different semantics. In certain cases, e.g because of the need to specify types, the left-hand-side will not even look like a valid expression. Following C. Böhm, we call this way of replacing <variables>s to the right of the '=' by what seem to be applied occurrences in argument postition to the left of it, the "heuristic application principle". Similar extensions allow the <type var> in a <polymorphic expr> and <pattern>s in a <lambda-case-of expr> to be moved to the left of '=' (it gets quite crowded there). We call the latter extension "pattern matching". Finally we also allow something analogous for <lambda-case-in expr>s.

Because all these extensions may apply in an interleaved order, we first give the syntax for <declaration>s, that will result from them all:

```
<simple declaration>
    ::= <formal> , <declaration body> .
<recursive declaration>
    ::= <typed formal> , <declaration body> .
<declaration body>
    ::= <declarative element> | <declaration body>+CHAIN"|" .
<declarative element>
    ::= <sample>* , "=" , <expression> .
<sample>
    ::= <formal> | "$",<type var> | <type>? , <pattern> | <integer denotation> .
```

The above syntax is deliberately ambiguous: to correctly parse a <declaration> it will be necessary to know the <constructor>s valid in the environment. We note that the <variable>s being declared by a <declaration> are all contained in the initial <formal>; all <variable>s and <type var>s appearing in the <sample>s are local to some <declaration body>.

First we give heuristic application for <lambda expr>s:

```
-> <declarative element> ; <sample>*,"=",<lambda expr>
:  <sample>*,"= `",<formal>,"->",<expression>
<- <sample>+,"=",<expression>
:  <sample>*,<formal>,"=",<expression>.
```

Similarly, we have heuristic application for <polymorphic expr>s (where "%" is replaced by "$" to make it look like a <specialisation>):

```
-> <declarative element> ; <sample>*,"=",<polymorphic expr>
:  <sample>*,"= %",<type var>,<expression>
<- <sample>+,"=",<expression>
:  <sample>*,"$",<type var>,"=",<expression>.
```

Finally we present pattern matching, which is a bit more complicated:

```
-> <declaration body> ; <declarative element>
;  <sample>*,"=",<lambda-case-of expr>
;  <sample>*,"= case",<type>?,"of",<constructor limb>i+CHAIN"|","esac"
:  <sample>*,"= case",<type>?,"of",(<pattern>i,"->",<expression>i)+CHAIN"|","esac"
<- <declaration body>i+CHAIN"|"
;  <declarative element>i+CHAIN"|"
:  <sample>*,<type>?, (<pattern>i,"=",<expression>i)+CHAIN"|".
```

Note that '<sample>*' and '<type>?' become part of the first <declarative element> only; the other <declarative element>s in the +CHAIN start off with a single (<type>-less) <pattern> (that may however be appended to by further extensions) which looks somewhat strange at first glance. We might have chosen to copy '<sample>*' and '<type>?' into each of the <declarative element>s, but that would only be useless writing effort to the programmer, and possibly tempt him to put something else there, which would be meaningless.

Concluding this group of extensions, we give the "pattern matching" for integers (always from 0 upwards):

```
-> <declaration body> ; <declarative element>
;  <sample>*,"=",<lambda-case-in expr>
:  <sample>*,"= case in",<expression>iLIST
             ,"out",<variable>,"->",<expression>0
             ,"esac"
<- <declaration body>i+CHAIN"|"
;  <declarative element>i+CHAIN"|"
:  <sample>*,(<integer denotation>i,"=",<expression>i)CHAIN"|"
       ,"|",<variable>,"=",<expression>).
```

The i-th <integer denotation> must represent the number (i-1) (we start from 0, and don't allow permutations here (why would we?)). Like with the ordinary pattern matching, the '<sample>*' becomes part of the first <declarative element> only. We give an example invloving all these possibilities:

```
'let @t ( (t|*) -> ( int -> int ) )
     f $t (succeed -) 0 = 3
                    | 1 = 2
                    | n = 2*n
        | (fail) x = x
```

```
 in E
'
stands for
'let @t ( (t|*) -> ( int -> int ) )
     f = %t case
           of (succeed -) -> case in 3,2 out n->2*n esac
           |  (fail)      -> 'x->x
           esac
 in E
'
```

It should also be clear by now how it can be determined how <declarative element>s should be grouped by the parser to form <declaration body>s, and how these must be grouped into bigger ones etc. Parsing a <declaration body>, the leftmost <sample> that is a '<type>?,<pattern>' or an <integer denotation>, not already accounted for by an enclosing <declaration body>, should be searched for. If it isn't found we have that this <declaration body> consists of a single <declarative element>. Otherwise we group together a '<declaration body>+CHAIN"|"' of which the other <declaration body>s start with <pattern>s or <integer denotation>s respectively (hereby accounted for). This grouping is finished when we have exhausted the <constructor>s remaining from the <constructor> in the <pattern> found initially, or respectively we reach a <declaration body> starting with a <variable> instead of a <integer denotation> (which last <declaration body> is included).

Finally we have heuristic application in <type declaration>s, similar to that of <polymorphic expr>s:

```
-> <type declaration> ; <type var>0,"=",<type generator>
:  <type var>0,"=",("%",<type var>i)+,<type>
<- <type var>0,("$",<type var>i)+,"=",<type>.
```

So we may write 'type pair$t = (t,t)' instead of 'type pair = %t (t,t)'.

II.3.8 Extensions For Lists

Until now we have not made any special provisions for lists. Indeed lists, that are so predominent in most other functional languages do not play a fundamental role in this one. We have not, however, forgotten them altogether, and do provide some minor sytactical conveniences specially for lists.

First of all, a <type generator> "list" is declared in the initial environment, together with constructors for it:

```
type list = %t rectype lt: (*|(t,lt)) in ...
constructors nil,cons for list in ...
```

Furthermore we provide special syntactic forms for applied uses of the constructors 'nil' and 'cons':

```
-> <enclosed expr> ; "(",<applied var>,")" : "(nil)"
<- <list display> : "(<>)".

-> <enclosed expr> ; "(",<function application>,")"
:  "(cons(",<expression>0,",",<expression>1,"))"
<- <cons form>
:  "(",<expression>0,":",<expression>1,")".

-> <enclosed expr> ; <cons form> : "(",<expression>,": nil$",<type>,")"
<- <list display> : "(<",<expression>,">)".
```

Here <type>=type(<expression>).

-> <enclosed expr> ; <cons form> ; "(",<expression>0,":",<list display>,")"
: "(",<expression>0,": (<",<expression>iLIST ,">))"
<- <list display> ; "(<",<expression>+LIST,">)"
: "(<",<expression>0,",",<expression>iLIST,">)".

So we have that e.g. '(<a,b>)' with a and b of type t, stands for '(cons(a,(cons(b,nil$t))))' Note that, like with array displays, the empty display '(<>)' is polymorphic, while the others aren't (and neither is any <cons form>). The strengths of the <expression>s occurring in a <cons form> (and consequently in a <list display>) equal the strength of that <cons form> (respectively <list display>), even though in the unextended construct they are weak (because the <expression>s are arguments to the polymorphic function 'cons'). In these particular cases the type propagation inward is easy (consequently, neither <cons form>s nor <list display>s (except '(<>)') are <coercend>s).

In <pattern>s we allow extensions similar to the first two above, but not for general "list display patterns" (there is no construct such a thing could reasonably be extended from).

-> <pattern> ; "(",<constructor>,")" : "(nil)" <- "(<>)".

-> <pattern> : "(cons(",var plan>0,",",<var plan>1,"))"
<- "(",<var plan>0,":",<var plan>1,")".

So 'case of (<>) -> A | (h:t) -> B esac' stands for
'case of (nil) -> A | (cons(h,t)) -> B esac'.

Naturally, all occurrences of 'nil' and 'cons' above are assumed to identify their defining occurrences in the initial environment.

II.3.9 Formulae

Our language would not be complete, of course, without allowing formulae (of a more general kind then those in the core language). It is generally recognised that formulae are semantically equivalent to (nested) function applications, presented in a different syntactical form, whose main advantages are compactness and symmetry of structure. Therefore formulae fit pefectly into our concept of extensions. Nevertheless formulae are such an intricate syntactical device, that they cannot be presented simply in our "rewrite" style for extensions. Therefore, in this section, we will take a "parsers view", telling how to deal with some given formula and bringing it back to <function application>s, rather than telling in which cases <function application>s may be specified as <formula>e.

Since we don't want to restrict the convenience of formulae to predefined operations on simple data types, we allow declaration of <operator>s. We need therefore consider the following points:

- lexical representations for <operator>s,
- operator- and priority-declarations,
- priority structure of <formula>e,
- operator identification,
- rewriting of <formula>e to <function application>s.

In general our handling of <formula>e will resemble, and is in fact for a great part shamelessly copied from, Algol 68 formulas.

A first classification of <formula>e is into monadic and dyadic ones; we have as a general form:

<formula> ::= <dyadic formula> | <monadic formula> .
<dyadic formula> ::= <operand> , <dyadic operator> , <operand> .
<monadic formula> ::= <monadic operator> , <operand> .

The last two rules are ambiguous and will be replaced by more precise ones.

II.3.9.1 Lexical Representation of Operators

Our first concern is lexical: <operator>s are represented by a sequence of one or more special "operator tokens" (or by <bold identifier>s), and since for sake of compactness we don't require adjacent <operator>s to be separated by a space (unless they are both <bold identifier>s), we need some restriction allowing an unambigous breaking up of a series of operator tokens into <operator>s. Since in a sequence of adjacent <operator>s, all but the first are necessarily <monadic operator>s, it is natural to distinguish operator tokens that may be used as (first token of) <monadic operator>s from those that may not. We call the former <monad>s, the latter <nomonad>s.

<monad> :== "+" | "-" | "~" | "#" | "!" | "?" .
<nomonad> :== "*" | "/" | "\" | "=" | "<" | ">" | "^" | "&" | "@" | "." .

The restriction is that a <monadic operator>s must start with a <monad>, and that any second or later token in an <operator> must be a <nomonad>:

<monadic operator> :== <monad> , <nomonad>* | <bold identifier> .
<dyadic operator> :== (<monad>|<nomonad>) , <nomonad>* | <bold identifier> .
<operator> :== (<monad>|<nomonad>) , <nomonad>* | <bold identifier> .

So we have that e.g. "*+", "+-", "=~", "?!" and "<->" are illegal as <operator>s (because their second tokens are <monad>s), "*", "*", "<=>" and ">>" may be used only as <dyadic operator>s (because they start with a <nomonad>) and "~", "-=>", "+*", and "max" are legal both as <dyadic operator> and as <monadic operator>.

The combination "->" is explicitly excluded from all three categories, since it is already used for other purposes. Also any <dyadic operator> starting with "." following an <integer denotation> must have a separation between them since otherwise there would be lexical ambiguity with a <real denotation>. We distinguish between monadic- and dyadic-operators only for occurrences in <formula>e, for other (defining) occurrences we only distinguish <operator>s, that may well be identified by both <monadic operator>s and <dyadic oprator>s (with above restriction that some symbols are only legal as <dyadic operator>).

II.3.9.2 Priorities

The next step is constructing a parse-tree for a (nested) <formula>; as usual this involves priorities. Prioities are used only for <dyadic operator>s, and are kept in the environment; an <operator> may not be used as <dyadic operator> unless a priority is attached to it in the environment. The initial environment defines some operator priorities, and priorities may be established (or changed) in a <priority declaration expr>:

<priority declaration expr>
    ::= "prio" , (<operator>,"=",<digit>)LIST , "in" , <expression> .

No <operator> should occur more than once in the LIST; the environment is extended in the obvious way with new operator priorites.

A <dyadic operator> that has priority n in the environment will be noted a <prio n operator>. The priority structure of <formula>e is now given by

```
<dyadic formula>
    ::= <prio 0 formula> | <prio 1 formula> | ... | <prio 9 formula> .
<prio 0 formula> ::= <prio 0 operand> , <prio 0 operator> , <prio 1 operand> .
<prio 1 formula> ::= <prio 1 operand> , <prio 1 operator> , <prio 2 operand> .
    .
    .
    .
<prio 9 formula> ::= <prio 9 operand> , <prio 9 operator> , <monadic operand> .
<monadic formula> ::= <monadic operator> , <monadic operand> .

<prio 0 operand> ::= <prio 0 formula> | <prio 1 operand> .
<prio 1 operand> ::= <prio 1 formula> | <prio 2 operand> .
    .
    .
    .
<prio 9 operand> ::= <prio 9 formula> | <monadic operand> .
<monadic operand> ::= <monadic formula> | <primary> .
```

The strength of any operand is weak. We see that there are 10 priority levels, labeled 0,1,..,9, that association within each level is to the left, and that monadic operators take precedence over all dyadic ones.

II.3.9.3 Operator Identification

Once the parse-tree has been constructed, the right algorithms should be associated with the operators. Since the core language gives a direct semantics only to <formula>e using the operators "descr" and "within", this calls for searching of the environment, called operator identification.

Some <operator>s occur in the initial environment, new ones and new uses of existing ones may be introduced in <var plan>s. For that purpose we define an extra production

```
<var plan> ::= ... | "op" , <operator> .
```

If such a <var plan> directly precedes the '=' of a <declaration>, it is required that there is a separation in between. Contrary to <variable>s, more than one defining occurrence of an <operator> may be accessible in an environment. Therefore we associate a set S of "operand types" to any occurrence of an <operator> in a <var plan>. The type of such a <var plan> must have the form '("@",<type var>i)*,"(",<type>0,"->",<type>1,")"', and any '<type var>i' present must occur in <type>0. Now S consists of all types obtainable by substituting any sequence of <type>s for the '<type var>i' into <type>0 (and corresponding result types may be obtained by the same substitution performed upon <type>1). We don't forbid one same <operator> occurring more than once in the same <formal>, but if this occurs the associated sets of operand types must be disjoint. (For the purpose of applying this rule, the <formal)s in one <declarations> should be considered as one, as they are in their unextended form.) The defining occurrences of the <operator>s, and their sets of operand types, are kept in the usual layerwise fashion in the environment.

Operator identification now proceeds as follows: to any <formula> an "operand type" T is associated. If the <formula> is a <monadic formula> of the form '<monadic operator>,<operand>', then T is 'type(<operand>)', if it is a <dyadic formula> of the form '<operand>1,<dyadic operator>,<operand>2', then T is '(type(<operand>1) , type(<operand>2))'. Now the <monadic operator> or the <dyadic operator> identifies the newest occurrence of that <operator> in the environment, such that T is in the set of operand types associated to that occurrence (the corresponding result type will be the type of the <formula>). If no such occurrence exists, the <formula> is syntactically incorrect.

Now we have defined the identification of applied and defining occurrences, we can rewrite a program so that it contains no more <formula>e, except the ones defined in the core language, as follows. We replace every <var plan> of the form '"op",<operator>' by a <variable> not used elsewhere, and rewrite any <formula> identifying that <operator> using an <applied var> identifying that <variable>: a <monadic formula> of the form '<monadic operator>,<operand>' is rewritten to '"(",<applied var>,"(",<operand>,"))"', while a <dyadic formula> of the form '<operand>1,<dyadic operator>,<operand>2' is rewritten to '"(",<applied var>,"(",<operand>1,",",<operand>2,"))"'; both forms are parenthesised <function application>s. Notice that we may have to change the <applied var> to a <specialisation> according to the forgetful extension presented earlier; this is always possible since <operand>s are always weak. A <dyadic formula> of the form '<operand>1,"within",<operand>2' where "within" identifies the occurrence of it in the initial environment, is rewritten to '"within","(",<operand>1,",",<operand>2,")"'.

II.3.9.4 Operator Specialisations

Although <operator>s will be used mostly in <formula>e, it might be desirable to incidentally use them in another position, e.g. as a parameter in a <function application>. Therefore we provide a syntactic form that allows operator identification without specifying operands.

<tertiary> ::= ... | <operator specialisation> .
<operator specialisation> ::= <operator>,"$",<type>0.

Operator identification proceeds as above with T equal to <type>0. The <operator specialisation> is replaced by the <applied var> identifying the <variable> replacing the <operator> identified, or if that <applied var> has a polymorphic type, by a (repeated) <specialisation> to it, that has a type of form '"(",<type>0,"->",<type>1,")"'.

II.3.10 Abstract Types

It is often desirable to create a <type var> and certain identifiers bound to functions, constants etc. related to it, and then restrict access to this <type var> to be limited to be solely via those identifiers. Using a <type declaration> won't suffice here, since a <type var> declared in this way stands for a <type> spelled out in full, allowing access to all operations possible according to the structure of that type. However a <type var> bound in a <polymorphic expr> (by '%'), does have the property that it displays no further structure (because it may stand for any type); the identifiers related to the <type var> may be introduced in a <lambda expr>. Now if we specialise the <polymorphic expr> to some concrete type, and then apply to a tuple of values giving the functions, constants etc. expressed using that concrete type, they may be used within the <polymorphic expr> without the concrete type being visible. We therefore present abstract type declarations as an extension for above combination of expressions; actually we allow more than one abstract type to be introduced at the same time.

```
-> <expression> ; <function application> ; <specialisation>,<expression>0
;  "(",<polymorphic expr>,")",("$",<type>i)+,<expression>0
;  "(",("%",<type var>i)+,<lambda expr>,")",("$",<type>i)+,<expression>0
:  "(",("%",<type var>i)+,"`",(<formal>|"(",<formal>j+LIST,")")
                          ,"->",<expression>1
  ,")",("$",<type>i)+,<expression>0
<- <abstract type declaration expr>
;  "abstype",<abstract type declaration>,"in",<expression>1
;  "abstype",<specification part>,"=",<implementation part>,"in",<expression>1
:  "abstype",<type var>iLIST,"with",(<formal>|<formal>j+LIST)
       ,"=",<type>iLIST,   ,"with",<expression>0
  ,"in",<expression>1.
```

We require that none of the <type var>i occur in 'type(<expression>1)'. So following "abstype" we give the abstract type(s) declared, then a (list of) <formal>(s) introducing <variable>s (or <operator>s) to be associated to the abstract type(s) (with their types being expressed in terms of the abstract type(s)). Than following the "=" comes the implementation part, first specifying the concrete type(s) instantiating the abstract one(s), then <expression>0 giving the implementation to be bound to the <formal>s. It will often be the case that <expression>0 is a <tuple display>, but it may also be a <let expr> declaring some auxiliary identifiers first, that will of course not be visible from within <expression>1. For an example of an <abstract type declaration>, see the declaration of complex numbers in the initial environment.

This extension demonstrates that the properties of explicit polymorphic typing allow abstract types to be described without any further mechanism. This way of defining abstract types is however somewhat limited, and has the following drawbacks, that cannot be remedied without changing the type system (which would go beyond the scope of extensions):

- We would like to have the <implementation part> act like a single expression, so that instead of having to give the concrete types right away, we might first do some declarations, especially of (abstract-)types. For this we would need some type to attach to the <implementation part> as a whole, which we do not have in our type system.
- We have forbidden that <expression>1, giving the yield of the whole <abstract type declararion expr> has any of the abstract types in its type. This was necessary, because the abstract types losing their validity, the concrete types would be substituted back for them (this is what happens in the unexended construct). This would be highly undesirable, but there is no other reasonable solution. However such exportation of abstract types would be very useful, if it could be used in specifying the <implementation part> of another abstract type, as was suggested above.
- We have no notion of polymorphic abstract types. Indeed it is rather useless putting an <abstract type declaration expr> inside a <polymorphic expr>, since the abstract type can be used only locally, and can't be exported to a point where the polymorphism might be specialised. Even nicer than to have the whole of the <abstract type declaration expr> be polymorphic, would it be to have just the <specification part> (and consequently of course also the <implementation part>) be parametrised by a type, defining a kind of "abstract type generator". In that case from within the scope of the abstract type declaration, we could specialise the abstract type generator, say "set_of", by any type X to obtain an abstract type, say "set_of$X". The operations related to the abtract type generator would of course also be polymorphic.

The first two mentioned points, and the third one partly, might be resolved by introducing a new type-quantifier (apart from '@'), say '&', to be interpreted as: "for some type, whose representation is suppressed here,...". Such an "existentially quantified" type could then be used to express the type we want an <implementation part> to have, allowing it to be treated as expression, without the concrete type being available to the type system. The properties of existentially

quantified types would then be that they are introduced by explicit acts of type-hiding, and may be used only in applications to sufficiently polymorhic functions (i.e. the type-correctness is verified without knowledge of the hidden type). This kind of use would include the use as <implementation part>. The result of an application of a polymorphic function to an object of existentially quantified type would again be existentially quantified, so that hiding of representation is preserved. This would (amongst others) allow (abstract) type declarations to be inserted at the beginning of an <implementation part>, and objects with hidden types to be yielded from abstract type declarations. Also it would allow a kind of polymorphism of abstract types, namely by a type of the form '@t1 &t2 T' it could be expressed that t2 may depend on t1. This would allow a polymorphic implementation part to be used in several abstract type declarations in different specialised instances. It would still not allow "abstract type generators", declared once and specialised at any needed use. In order to achieve this we would need an even more drastic extension of the type system, namely allowing type-quantification to range not only over single types, but also over type generators of any arity. It is not at all evident what the consequences of such an extension would be for the integrity and complexity of the type system.

In any case incorporating these suggestions into the language at this stage would clearly have been premature, and also would go beyond the possibilties of extensions in the sense used here.

II.4 INITIAL ENVIRONMENT

In this section we give the set of <identifier>s, <type var>s, <operator>s and priorities that are predefined in the language, so that they can be used in any program without need of declaration. They will be give partly by sample declarations of various kinds, ending with 'in ...' to indicate that the further declarations, or, in case of the last declaration, the user-program, are assumed to be in the scope of this declaration.

II.4.1 Type Declarations

```
type pair$t = (t,t) in
type bool = (*|*) in
type predicate$t = (t->bool) in    type relation$t = predicate$pair$t in
type com = (*|*|*) in    type compar$t = (pair$t->com) in
type string = []char in
type list$t = rectype lt: (*|(t,lt)) in
type unop$t = (t->t) in    type binop$t = (pair$t->t) in
type field$t = ( binop$t , binop$t {addition,subtraction}
               , binop$t , binop$t {multiplication,division}
               , unop$t , unop$t {negation,inversion}
               , predicate$t {equality}
               , t {zero} , t {one} , int {characteristic}
               ) in ...
```

II.4.2 Constructor Declarations

```
constructors true,false for bool in
constructors less,equal,greater for com in
constructors minus,zero,plus for %t(t|*|t) in
constructors stop,include for %t%s(*|(t,s)) {for cumulate} in
constructors nil,cons for list in ...
```

II.4.3 Built-in Functions

We give identifiers for all built-in functions, their types, and a description of the algorithm they represent. All these algorithms are assumed to be primitively present in te core language. In the terminology of reduction systems, they are called delta-rules. Where an <expression> appears as description, this is meant for clarification of the desired extensional behaviour only, not to indicate that this <expression> should be actually used to perform the algorithm. Comments appearing in <expression>s indicate a value in ordinary mathematical notation.

```
identifier      type                        description
-------------------------------------------------------------------------------
not             (bool->bool)                `x->if x then false else true fi
bool_eq         ((bool,bool)->bool)         `(x,y)->if x then y else not y fi
bool_ne         ((bool,bool)->bool)         `x->not(bool_eq x)
bool_abs        (bool->int)                 `x->if x then 1 else 0 fi

int_lt          ((int,int)->bool)           `(x,y)-> { x<y }
int_gt          ((int,int)->bool)           `(x,y)-> int_lt(y,x)
int_eq          ((int,int)->bool)           `(x,y)-> { x=y }
int_ge          ((int,int)->bool)           `(x,y)-> not(int_lt(x,y))
int_le          ((int,int)->bool)           `(x,y)-> not(int_lt(y,x))
int_ne          ((int,int)->bool)           `(x,y)-> not(int_eq(x,y))
int_negate      (int->int)                  `x-> { -x }
succ            (int->int)                  `x-> { x+1 }
pred            (int->int)                  `x-> { x-1 }
int_add         ((int,int)->int)            `(x,y)-> { x+y }
int_sub         ((int,int)->int)            `(x,y)-> { x-y }
int_mul         ((int,int)->int)            `(x,y)-> { x*y }
int_sign_abs    (int->(int|*|int))          `x->
                                            if int_lt(x,0) then minus(int_negate x)
                                            elif int_eq(x,0) then zero
                                            else plus x
                                            fi
div_mod         ((int,int)->(int,int))      rec f: `(x,y)->
                                            case int_sign_abs y
                                            of (minus yy) ->
                                                let (q,r)=f(x,yy) in (int_negate q,r)
                                            | (zero) -> error "Integer divide by 0"
                                            | (plus y)  ->
                                                if int_lt(x,0) then
                                                    let (q,r)=f(int_add(x,y),y)
                                                    in (pred q,r)
                                                elif int_lt(x,y) then (0,x)
                                                else
                                                    let (q,r)=f(int_sub(x,y),y)
                                                    in (succ q,r)
                                                fi
                                            esac
div_2           (int->(int,bool))           `x ->
                                            let (q,r)=div_mod(x,2) in (q,int_eq(r,1))
mul_2           ((int,bool)->int)           `(n,p)-> int_add(int_mul(2,n),bool_abs p)
int_power       ((int,int)->int)            rec f: `(x,y) ->
                                            case int_sign_abs y
                                            of (minus -) -> error "Negative exponent"
                                            | (zero) -> 1
                                            | (plus y) -> int_mul(x,f(x,pred y))
                                            esac
```

```
shift           ((int,int)->int)        `(k,n)->
                                        case int_sign_abs k
                                        of (minus kk) ->
                                            let (q,-)=div_mod(n,int_power(2,kk)) in q
                                        | (zero) -> n
                                        | (plus n) -> int_mul(n,int_power(2,k))
                                        esac

float           (int->real)             { The embedding function }
real_lt         ((real,real)->bool)     `(x,y)-> { x<y }
real_gt         ((real,real)->bool)     `(x,y)-> real_lt(y,x)
real_eq         ((real,real)->bool)     `(x,y)-> { x=y }
real_ge         ((real,real)->bool)     `(x,y)-> not(real_lt(x,y))
real_le         ((real,real)->bool)     `(x,y)-> not(real_lt(y,x))
real_ne         ((real,real)->bool)     `(x,y)-> not(real_eq(x,y))
real_negate     (real->real)            `x-> { -x }
real_invert     (real->real)            `x-> { 1/x }
real_add        ((real,real)->real)     `(x,y)-> { x+y }
real_sub        ((real,real)->real)     `(x,y)-> { x-y }
real_mul        ((real,real)->real)     `(x,y)-> { x*y }
real_div        ((real,real)->real)     `(x,y)-> { x/y }
entier          (real->int)             `x-> { The largest integer n
                                               such that n<=x }
round           (real->int)             `x-> entier(real_add(x,0.5))
real_sign_abs   (real->(real|*|real))   `x->
                                        if real_lt(x,0) then minus(real_negate x)
                                        elif real_eq(x,0) then zero
                                        else plus x
                                        fi
real_power      ((real,int)->real)      rec f: `(x,y) ->
                                        case int_sign_abs y
                                        of (minus yy) -> real_invert(f(x,yy))
                                        | (zero) -> 1
                                        | (plus y) -> real_mul(x,f(x,pred y))
                                        esac

pi              real                    { The best approximation of pi }
sqrt            (real->real)            { The square-root function }
ln              (real->real)            { The natural logarithm function }
exp             (real->real)            { The inverse function of ln }
sin             (real->real)            { The sine function }
cos             (real->real)            { The cosine function }
tan             (real->real)            { The tangent function }
arcsin          (real->real)            { The inverse sine function }
arccos          (real->real)            `x ->
                                        real_sub(real_div(pi,2.),arcsin(x))
arctan          (real->real)            { The inverse tangent function }
                                        N.B. Both arcsin and arctan yield result
                                        values between -pi/2 and pi/2.
next_random     (int->(int,real))       `n ->
                                        ( { The integer following n in some
                                            pseudo-random sequence }
                                        , { The real value r with 0<=r<1
                                            obtained by applying some uniform
                                            mapping to the integer computed }
                                        )

ascii_value     (char->int)             `c -> { The integer representing c
                                                in the ASCII code }
ascii_char      (int->char)             { The partial inverse of ascii_value }

split           @t(([]t->int)->([]t->([]t,[]t))) { See section II.2.6.5 }
```

```
concatenate    @t(([]t,[]t)->[]t)                { See section II.2.6.5 }
fold           @t@s((((t,s)->s),s)->([]t->s))    { See section II.2.6.5 }
cumulate       @t@s((s->(*|(t,s)))->(s->[]t))    { See section II.2.6.5 }
select         @t((t->bool)->([]t->[]t))         { See section II.2.6.5 }
random_write   @t([](int,t)->[]t)                { See section II.2.6.5 }
```

II.4.4 Priority Declarations

```
prio => = 0 , <=> = 0
, or = 1
, and = 2 , & = 2
, = = 3 , /= = 3 , ~= = 3 , # = 3 , xor = 3
, < = 4 , > = 4 , <= = 4 , >= = 4 , cmp = 4 , within = 4
, - = 5 , + = 5
, / = 6 , \ = 6 , div = 6 , /* = 6 , mod = 6 , \* = 6
, * = 7
, ^ = 8 , ** = 8
, . = 9 , +* = 9 , i = 9 , ? = 9 , ! = 9
in ...
```

II.4.5 Initial Declarations

The following (most operator-) declarations are not part of the core language, and the expressions appearing may be used in replacing a program by an equivalent one in the core language. Therefore, they do not define new built-in functions.

```
let double$t t x = (x,x) ,  triple$t t x = (x,x,x)
; id$t t x = x
, (const,op k) = double( %t `t x -> %s `s - -> x )
, (compose,op .) = double( %a%b%c `( (b->c)f , (a->b)g ) -> `a x -> f(g(x)) )
, 1_of_2$a$b (a x,b -) = x , 2_of_2$a$b (a -,b y) = y

; (op not,op ~) = double not
, (op =,op <=>) = double bool_eq
, ((op /=,op ~=),(op #,op xor)) = double(double bool_ne)
, op or  pair$bool(p,q) = if p then true else q fi
, (op and, op &) = double( `pair$bool(p,q) -> if p then q else false fi )
, (op implies, op =>) = double( `pair$bool(p,q) -> if p then q else true fi )
, op abs = bool_abs
, op < = int_lt , op > = int_gt , op <= = int_le , op >= = int_ge
, op = = int_eq , (op /=,op ~=,op #) = triple int_ne
, op - = int_negate , op + = id$int
, op + = int_add , op - = int_sub , op * = int_mul
, op / pair$int(n,m) = real_div(float n,float m)
, op # = int_sign_abs , op \* = div_mod
; op abs  int n = case #n of nn->nn | - ->0 | n->n esac
, op sign int n = case #n of - ->-1 | - ->0 | - ->1 esac
, (op \,op div) = double( `pair$int p -> 1_of_2(div_mod p) )
, (op /*,op mod) = double( `pair$int p -> 2_of_2(div_mod p) )
, op half int n = 1_of_2(div_2 n) , op odd  int n = 2_of_2(div_2 n)
, (op ^,op **) = double int_power
, op min pair$int(n,m) = if n<m then n else m fi
, op max pair$int(n,m) = if n<m then m else n fi
, (int->(int->[]int)) op base b = { convert integer to row of digits }
    cumulate case in stop out n -> let (q,r)=n\*b in include(r,q) esac
, (int->([]int->int)) op radix b = { the inverse of base b }
    fold ( `pair$int(r,q) -> r+b*q , 0 )
```

```
, op < = real_lt , op > = real_gt , op <= = real_le , op >= = real_ge
, field$real real_field==(op +,op -,op *,op /,op -,op inv,op =,-,-,-)
  = (real_add,real_sub,real_mul,real_div,real_negate,real_invert,real_eq,0.,1.,0)
, (op /=,op ~=,op #) = triple real_ne
, op + = id$real
, op # = real_sign_abs
; op abs  real x = case #x of xx->xx | - ->0. | x->x esac
, op sign real x = case #x of - ->-1 | - ->0  | - ->1 esac
, (op ^,op **) = double real_power
, op round = round , op entier = entier , op fl = float
, op min pair$real(n,m) = if n<m then n else m fi
, op max pair$real(n,m) = if n<m then m else n fi

, op lwb $t  []t r = 1_of_2(descr r)
, op upb $t  []t r = 2_of_2(descr r)
, op size $t []t r = let (l,u)=descr r in u+1-l
, halfway$t  []t r = let (l,u)=descr r in half(l+u) {use with split}
, op ~ $t []t r = r<[~]>
, op + = concatenate
; op + $t ([]t r,t a) = r+[[a]] , op + $t (t a,[]t r) = [[a]]+r

; compar$int op cmp  (n,m) = if n<m then less elif n=m then equal else greater fi
, compar$real op cmp (n,m) = if n<m then less elif n=m then equal else greater fi
; compar$char op cmp (n,m) = (ascii_value n)cmp(ascii_value m)
; compar$string op cmp (s,t) =
    let (s,t) = (s<[at 1]>,t<[at 1]>)
    ; min_u = min(upb s,upb t)
    ; rec (int->com)
      f i = if i>min_u then equal
            else case s[i] cmp t[i]
                 of (less)->less | (equal)->f(succ i) | (greater)->greater
                 esac
            fi
    in case f 1
       of (less)->less | (equal)->upb s cmp upb t | (greater)->greater
       esac

, op abs  com(less) = -1
          | (equal) = 0
          | (greater) = 1
; @t(compar->%r(r,r,r,r,r,r)$relation$t) compar_to_relations$t op cmp =
    let op ? pair$t p = abs cmp p
    in (`p->?p=0 , `p->?p~=0 , `p->?p<0 , `p->?p>0 , `p->?p<=0 , `p->?p>=0)
; (op =,op ~=,op <,op >,op <=,op >=) = compar_to_relations(cmp$pair$char)
, (op =,op ~=,op <,op >,op <=,op >=) = compar_to_relations(cmp$pair$string)
, ((op /=,op /=),(op #,op #)) = double(~=$pair$char,~=$pair$string)

, (reverse,op ~) = double
    ( %t let rec binop$list$t
             siphon (a,b) = case a of (<>) -> b | (h:t) -> siphon(t,(h:b)) esac
        in `list$t l -> siphon (l,(<>))
    )
, (fold_left,op /) = double
    ( %s%t `( s start , ((s,t)->s)op ? ) ->
          prio <== = 4 in
          let rec ((s,list$t)->s)
              op <== (val,l) = { accumulate elements of l into val }
                 case l of (<>) -> val | (h:t) -> val?h <== t esac
          in `list$t l -> start <== l
    )
, (fold_right,op \) = double
    ( %t%s `( ((t,s)->s)op ? , s start ) ->
```

```
            rec (list$t->s) fold:
            case of (<>) start | (h:t) -> h?(fold t) esac
    )
, (append,op +) = double
    ( %t rec binop$list$t op +:
        `(a,b) -> case a of (<>) -> b | (h:t) -> (h:t+b) esac
    )
, (map_list,op !) = double
    ( %s%t `(s->t)f ->
            rec (list$s->list$t) map_f:
            case of (<>) -> (<>) | (h:t) -> (f h:map_f t) esac
    )
in
abstype compl
with (pair$real->compl) cart_to_compl , (compl->pair$real) compl_to_cart
   , (pair$real->compl) polar_to_compl , (compl->pair$real) compl_to_polar
   , field$compl complex_field
     ==(op +,op -,op *,op /,op -,op inv,op =,-,-,-)
   , unop$compl complex_conjugate

= pair$real  {real and imaginary part}
with ({cartesian conversions are identity:} id, id
      {polar conversions:}
     ,`(r,phi) -> (r*(cos phi),r*(sin phi))
     ,`(re,im)
      -> ( sqrt(re*re+im*im)
         , if re=0. & im=0. then 0.
           elif abs re<=abs im then (pi/2.)*fl sign im-(arctan(re/im))
           else let x = arctan(im/re)
                in if re>0. then x elif im>=0. then x+pi else x-pi fi
           fi
         )
      {field operations:}
     ,( {+} `((a,b),(c,d)) -> (a+c,b+d) , {-} `((a,b),(c,d)) -> (a-c,b-d)
      , {*} `((a,b),(c,d)) -> (a*c-b*d,b*c+a*d)
      , {/} `((a,b),(c,d)) -> let r2=c*c+d*d in ((a*c+b*d)/r2,(b*c-a*d)/r2)
      , {-} `(a,b) -> (-a,-b) , {inv} `(a,b) -> let r2=a*a+b*b in (a/r2,-b/r2)
      , {=} `((a,b),(c,d)) -> a=c & b=d
      , {zero} (0.,0.) , {one} (1.,0.) , {characteristic} 0
      )
      {complex conjugation:}
     ,`(a,b) -> (a,-b)
     )
in
let (op +*,op i) = double cart_to_compl , op # = compl_to_cart
, op ? = polar_to_compl , op ! = compl_to_polar
; op re = `compl z -> 1_of_2(#z), op im = `compl z -> 2_of_2(#z)
, op abs = `compl z -> 1_of_2(!z), op arg = `compl z -> 2_of_2(!z)
, op + = id$compl , op ~ = complex_conjugate, op co = `real x -> x+*0.
, (op /=,op ~=,op #) = triple (not.(=$pair$compl))
; (op ^,op **) = double
    ( rec ((compl,int)->compl) op ^: `(z,n) ->
      case #n
      of (minus nn) -> inv(z^nn)
      | (zero) -> co 1.
      | (plus n) -> let (half_n,odd) = div_2 n ; w = z^half_n
                    in if odd then w*w*z else w*w fi
      esac
    )

in ...
```

II.5 SYNTAX

In this section we present the collected syntax of the language. We separate the rules of the (context-free) grammar, and those giving the lexical representations of the symbols. Occasionally a new syntactical category will be introduced, but this is only done to clarify the syntactical structure.

II.5.1 Grammar

```
<expression>
    ::= <lambda expr> | <recursion expr> | <polymorphic expr> | <let expr>
      | <type declaration expr> | <constructor declaration expr>
      | <priority declaration expr> | <abstract type declaration expr>
      | <tertiary> .
<tertiary>
    ::= <function application> | <where expr> | <operator specialisation>
      | <secondary> .
<secondary>
    ::= <formula> | <primary> .
<primary>
    ::= <specialisation> | <subscription> | <array update>
      | <descriptor transformation> | <applied var> | <denotation>
      | <error expr> | <enclosed expr> .
<enclosed expr>
    ::= <tuple display> | <union display> | <array display> | <list display>
      | <case-of expr> | <lambda-case-of expr> | <conditional>
      | <case-in expr> | <lambda-case-in expr> | <cons form>
      | <for expr> | <tabulation expr> | "(" , <expression> , ")" .

<type>
    ::= <function type> | <recursion type> | <polymorphic type>
      | <primitive type> | <tuple type> | <union type> | <array type>
      | <type generator> , ( "$" , <type> )+ .
<function type>
    ::= "(" , <type> , "->" , <type> , ")" .
<recursion type>
    ::= "rectype" , <type var> , ":" , <type> .
<polymorphic type>
    ::= "@" , <type var> , <type> .
<primitive type>
    ::= <base type> | <type var> .
<base type>
    ::= "char" | "int" | "real" .
<tuple type>
    ::= "*" | "(" , <type>+LIST , ")" .
<union type>
    ::= "(" , <type>+CHAIN"|" , ")" .
<array type>
    ::= "[" , <empty>LIST , "]" , <type> .
<type generator>
    ::= ( "%" , <type var> )+ , <type> | <type var> .

<lambda expr>
    ::= "\" , <formal>+ , ( "." | "->" ) , <expression> .
<formal>
    ::= <typed formal> | <var plan> .
<typed formal>
    ::= <type> , <var plan> | "(" , <typed formal>+LIST , ")" | "()" .
<var plan>
    ::= <variable> | "op" , <operator> | <compound plan>
      | <variable> , "==" , <compound plan> | "-" .
```

```
<compound plan>
    ::= "(" , <var plan>+LIST , ")" | "()" .

<recursion expr>
    ::= "rec" , <formal> , ":" , <expression> .
<polymorphic expr>
    ::= "%" , <type var> , <expression> .

<let expr>
    ::= "let" , <declarations>CHAIN";" , "in" , <expression> .
<where expr>
    ::= <tertiary> , "where" , <declarations> , "end" .
<declarations>
    ::= <simple declaration>LIST | "rec" , <recursive declaration>LIST .
<simple declaration>
    ::= <formal> , <declaration body> .
<recursive declaration>
    ::= <typed formal> , <declaration body> .
<declaration body>
    ::= <declarative element> | <declaration body>CHAIN"|" .
<declarative element>
    ::= <sample>* , "=" , <expression> .
<sample>
    ::= <formal> | "$" , <type var> | <type>? , <pattern> | <integer denotation> .
<pattern>
    ::= "(" , <constructor> , <var plan>? , ")"
      | "(<>)" | "(" , <var plan> , ":" , <var plan> , ")" .

<type declaration expr>
    ::= "type" , <type declaration> , "in" , <expression> .
<constructor declaration expr>
    ::= "constructors" , <constructor declaration> , "in" , <expression> .
<priority declaration expr>
    ::= "prio" , <priority declaration> , "in" , <expression> .
<abstract type declaration expr>
    ::= "abstype" , <abstract type declaration> , "in" , <expression> .

<type declaration>
    ::= <type var> , "=" , ( <type> | <type generator> )
      | <type var> , ( "$" , <type var> )+ , "=" , <type> .
<constructor declaration>
    ::= <variable>+LIST , "for" , ( <type> | <type generator> ) .
<priority declaration>
    ::= ( <operator> , "=" , <digit> )LIST .
<abstract type declaration>
    ::= <specification part> , "=" , <implementation part> .
<specification part>
    ::= <type var>LIST , "with" , <formal>LIST .
<implementation part>
    ::= <type>LIST , "with" , <expression> .

<function application>
    ::= <tertiary> , <primary> .
<operator specialisation>
    ::= <operator> , "$" , <type> .
<formula>
    ::= <dyadic formula> | <monadic formula> .
<dyadic formula>
    ::= <prio 0 formula> | <prio 1 formula> | ... | <prio 9 formula> .
<prio 0 formula>
    ::= <prio 0 operand> , <prio 0 operator> , <prio 1 operand> .
```

```
<prio 1 formula>
    ::= <prio 1 operand> , <prio 1 operator> , <prio 2 operand> .
    .
    .
    .
<prio 9 formula>
    ::= <prio 9 operand> , <prio 9 operator> , <monadic operand> .
<monadic formula>
    ::= <monadic operator> , <monadic operand> .
<prio 0 operand>
    ::= <prio 0 formula> | <prio 1 operand> .
<prio 1 operand>
    ::= <prio 1 formula> | <prio 2 operand> .
    .
    .
    .
<prio 9 operand>
    ::= <prio 9 formula> | <monadic operand> .
<monadic operand>
    ::= <monadic formula> | <primary> .

<specialisation>
    ::= <primary> , "$" , <type> .
<subscription>
    ::= <primary> , "[" , <expression>LIST , ( "ext" , <expression> )? , "]" .
<array update>
    ::= <component update> | <exchange> .
<component update>
    ::= <primary> , "([" , <expression>LIST , "]:=" , <expression> , ")" .
<exchange>
    ::= <primary>
      , "(["    , ( <expression> | <empty> )LIST
      , "]<->[" , ( <expression> | <empty> )LIST
      , "])" .
<descriptor transformation>
    ::= <primary> , <modifier> .
<modifier>
    ::= <paster> | <merged modifier> .
<paster>
    ::= "<[" , (<empty>LIST)+CHAIN"|" , "]>" .
<merged modifier>
    ::= "<[" , ( <integer denotation>? , <trim>* )CHAIN("," | "][") , "]>" .
<trim>
    ::= "~" | ( ";" | ":" | "at" | ) , <expression> .

<denotation>
    ::= <character denotation> | <integer denotation> | <real denotation> .
<error expr>
    ::= "error" , <enclosed expr> .

<tuple display>
    ::= "(" , <expression>+LIST , ")" .
<union display>
    ::= "(" , <expression> , ( "|" , <type>? )+ , ")"
      | "(" , ( <type>? , "|" )+ , <expression> , ( "|" , <type>? )* , ")" .
<array display>
    ::= "[[]]" | "[[" , <expression>LIST , "]]" | <string> .
<list display>
    ::= "(<>)" | "(<" , <expression>LIST , ">)" .
```

```
<case-of expr>
    ::= "case" , <expression> , "of" , <caselimbs> , "esac" .
<lambda-case-of expr>
    ::= "case" , <type>? ,"of" , <caselimbs> , "esac" .
<caselimbs>
    ::= <limb>CHAIN"|" | <constructor limb>CHAIN"|" .
<limb>
    ::= <expression> | <var plan> , "->" , <expression> .
<constructor limb>
    ::= <pattern> , "->" , <expression> .
<conditional>
    ::= "if" , <expression> , "then" , <expression> , <else part> .
<else part>
    ::= "elif" , <expression> , "then" , <expression> , <else part> .
      | "else" , <expression> , "fi" .
<case-in expr>
    ::= "case" , <expression> , "in" , <expression>LIST , "out" , <limb> , "esac" .
<lambda-case-in expr>
    ::= "case in" , <expression>LIST , "out" , <limb> , "esac" .

<cons form>
    ::= "(" , <expression> , ":" , <expression> , ")" .
<for expr>
    ::= "for" , <generator>LIST , ":" , <limb> , "rof" .
<generator>
    ::= <expression>CHAIN"||" .
<tabulation expr>
    ::= "tab" , <expression> , ":" , <limb> , "bat" .
```

II.5.2 Lexical Representation

```
<empty>
    :== .
<letter>
    :== "a" | "b" | ... | "z" | "_" .
<digit>
    :== "0" | "1" | ... | "9" .
<bold letter>
    :== "a" | "b" | ... | "z .
<monad>
    :== "+" | "-" | "~" | "#" | "!" | "?" .
<nomonad>
    :== "*" | "/" | "\" | "=" | "<" | ">" | "^" | "&" | "@" | "." .

<character denotation>
    :== "'" , <ascii character> .
<integer denotation>
    :== <digit>+ .
<real denotation>
    :== <digit>+ , "." , <digit>* , ( "e" , "-"? , <digit>+ )? .

<variable>
    :== <identifier> .
<applied var>
    :== <identifier> .
<constructor>
    :== <identifier> .
<identifier>
    :== <digit>* , <letter> , ( <letter> | <digit> )* .
```

```
<type var>
    :== <bold identifier> .
<bold identifier>
    :== <bold letter>+ .

<operator>
    :== ( <monad> | <nomonad> ) , <nomonad>* | <bold identifier> .
<monadic operator>
    :== <monad> , <nomonad>* | <bold identifier> .
<dyadic operator>
    :== ( <monad> | <nomonad> ) , <nomonad>* | <bold identifier> .

N.B. "->" is not a legal <operator>, <monadic operator> or <dyadic operator>.

<comment>
    :== "{" , (<any ascii character but curly brackets>|<comment>)* , "}" .
<string>
    :== <quote> , <string item>* , <quote> .
<string item>
    :== <any ascii character but quote> | <quote image> .
<quote image>
    :== <quote> , <quote> .
<quote>
    :== """ . { a single copy of ascii character number 34 ! }
```

References

L. Augustsson
[1984] A compiler for lazy ML, proc. ACM conf. on LISP and functional programming, 218-227.

J. Backus
[1978] Can programming be liberated from the von Neumann style? Comm. ACM 21(4), 613-641.

H.P. Barendregt
[1984] The lambda calculus, its syntax and semantics. North Holland, Amsterdam.

Burstall, R.M., D.B. MacQueen and D.T. Sannella
[1980] HOPE: an experimental applicative language, in: Proceedings first LISP conference, Stanford, 136-143.

N.G. de Bruijn
[1972] Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, Indag. Math. 34, 381-392.

J.-Y. Girard
[1972] Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur, Ph.D. thesis, Université de Paris VII.

T. Johnsson
[1984] Efficient compilation of lazy evaluation, proc. of 1984 ACM SIGPLAN conf. on compiler constr., Montreal.

G.Kahn, D.B. MacQueen and G. Plotkin (eds.)

[1984] Semantics of data types, LNCS 173, Springer, Heidelberg.

J.W. Klop
[1980] Combinatory reduction systems, Mathematical Center Tracts, Kruislaan 413, 1098 SJ Amsterdam.

P. Landin
[1964] The mechanical evaluation of expressions, Computer Journal 6, 308-320.
[1965] A correspondence between Algol 60 and Church's lambda notation, Comm.ACM, 8(2), 89-101, 158-165.
[1966] A lambda calculus approach, in: Advances in programming and nonnumerical computation (L. Fox ed.), Pergamon Press, Oxford, 97-141.
[1966a] The next 700 programming languages, Comm. ACM, 9,157-166.

D. MacQueen, G. Plotkin and R. Sethi
[1984] An ideal model for recursive polymorphic types, Eleventh Annual ACM Symposium on Principles of Programming Languages, January 1984, Salt Lake City, Utah.

J.C.Reynolds
[1974] Towards a theory of type structure, proc. Colloque sur la programmation, LNCS 19, Springer, Heidelberg, 408-425.

D. Scott
[1976] Data types as lattices, Siam J. of comput. 5, 522-587.

D. Turner
[1979] A new implementation technique for applicative languages, Software prctice and experience 9, 31-49.
[1979a] SASL language manual, preprint obtainable at Computer Laboratory, University of Kent, Canterburry, England.
[1985] Miranda: a non-strict functional language with polymorphic types, in: Functional programming languages and computer architecture (Ed. J.-P. Jouannaod), Lecture Notes Computer Science 101, Springer, Heidelberg, 1-16.

Section III will then propose an adapted design methodology for building a hierarchy of protocols . It gives the most important design aspects which have to be addressed during the design: architecture, modelling and verification.

The last section gives a partial example of an important and complex protocol, the protocol defined for the Transport layer. Part of the Transport class 2 Protocol will be discussed .

II - LAYERS IN PROTOCOLS

It is now clear to protocol experts that the complexity of important distributed systems cannot be managed without using the layering principle. The corresponding approach and architecturing principle is given in /ZIM/. Most of the work in complex open protocols refer to the OSI architectural model, a reference model which is a standard for ISO and CCITT.

II.1 - Protocols and services

The reel aim of layering is structuring complexity, structuring in the sense that in order to design a layer it should be not necessary to fully know and understand the set of all the layers that exist under the considered layer.

As the lower layers are composed of at least one protocol per layer, dealing with for instance five lower layers means to deal with at least five protocols that implement the lower layer behaviors. In order to avoid for the designers the need of fully understand the resulting complexity, the concept of service has been introduced:see for instance /VIS/.

The aim of the service concept is to define how the lower layers act when they are used by a considered layer which is located on top of them.The service must allow the hiding of everything -in the lower layers- which is unneeded for the design of the considered layer.

In other words: on one hand, a service defines what are the functions offered to a given layer by the layers under it and how these layers are seen from the considered layer; on the other hand, a protocol defines how the functions of a given service are actually realized, i.e. by which real exchanges of messages. In other words, a service defines a global function,